# Java Message Service

CARA M2 Miage FA-FC

Jean-François Roos - bâtiment M3 extension - bureau 218 - Jean-Francois.Roos@univ-lille1.fr

6 mars 2016

# Sommaire

1 Introduction

2 Concepts de base JMS

3 Le modèle de programmation

4 Une application cliente

5 JMS dans une application JEE

# Message Oriented Middleware

- Les MOM sont des systèmes fournissant à leurs clients un service Peer to Peer : c'est un logiciel serveur dont le rôle est de fédérer l'envoi et la réception de ces messages entre les différents types d'applications.
- La communication de deux applications via un MOM est complètement asynchrone, c'est à dire que l'émetteur et le destinataire n'ont pas besoin d'être connectés simultanément lorsqu'ils communiquent.
- La communication n'est synchrone qu'entre l'émetteur et le MOM d'une part, et le MOM et le destinataire d'autre part.

# Mode de fonctionnement

- Utilisation de files d'attentes ou queues par lesquelles transitent les messages : lorsqu'un applicatif envoie un message, il se connecte au broker de messages (courtier de messages) à qui il envoie le message en précisant l'identifiant de la file d'attente.
- Quand le destinataire du message se connecte au broker, le message lui est délivré lorsqu'il lit la file d'attente en question.
- Une file d'attente peut aussi être utilisée pour plusieurs couples d'applicatifs (pas besoin de dédier une file par liaison applicative) puisque les MOM comportent différents critères de sélection de messages lors de la lecture.
- Par ailleurs, comme c'est le cas pour une table d'une base de données, les messages peuvent être aussi consultés sans être lus, c'est ce qu'on appelle le mode "browse".

# Définitions

- **MOM store and forward** : le MOM stocke le message et le route par la suite à son destinataire lorsque celui-ci le demande.
- **Provider** : produit, logiciel, qui fournit des services d'envoi/réception de messages en mode asynchrone.
- **Message** : informations échangées ; trois parties distinctes :
  - les données elles mêmes
  - l'entête du message (son identifiant, date de dépot ...)
  - ses propriétés, les caractéristiques fonctionnelles du message, qui sont différentes pour chaque application émettrice
- **Queue** : espace de stockage pour les messages, limité ou non.

# Définitions

- **Topic** : file d'attente particulière destinée à recevoir les messages utilisés dans le mode Publish/Subscribe
- **Queue Manager** : ensemble de files d'attentes formant un tout cohérent et géré par une ou plusieurs instances d'un MOM. Le queue manager est l'équivalent dans le monde des MOM d'une base de données dans le monde des SGBD.
- **Channel** : lien entre deux queue manager qui communiquent et s'envoient des messages.
- **Nœud** : lors de la communication entre deux gestionnaires de files d'attente par le biais d'un canal, on parle de communication entre deux nœuds.
- **Routage**

# Sommaire

1 Introduction

2 Concepts de base JMS
- Architecture JMS
- Messaging Domains
- Récupération des messages

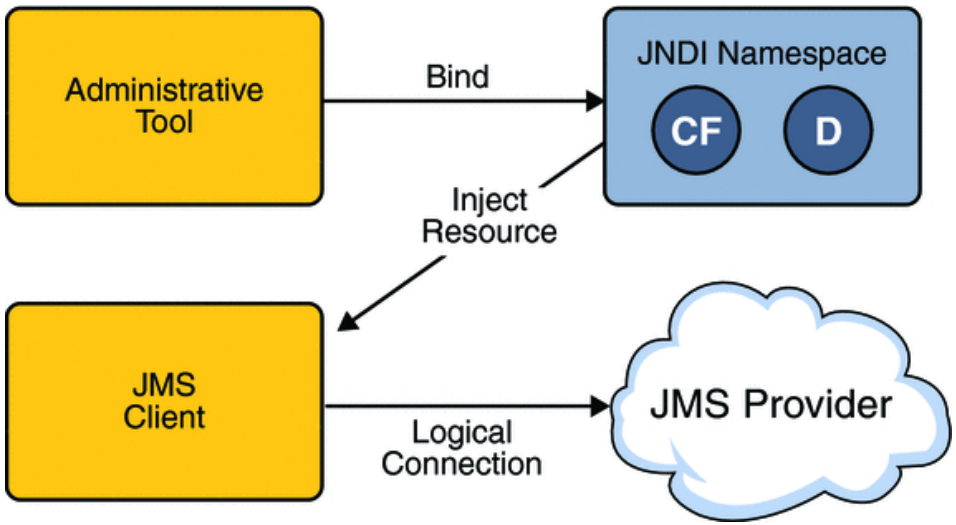
3 Le modèle de programmation

4 Une application cliente

5 JMS dans une application JEE

- Un provider JMS est un système à message qui implémente les interfaces JMS et qui fournit des fonctionnalités d'administration et de contrôle. Une implémentation de JEE inclue un provider JMS.
- Les clients JMS sont des composants écrits en Java qui produisent et consomment des messages. Tout composant JEE peut être client JMS.
- Les messages sont les objets échangés entreles clients JMS.
- Les objets administrés sont des objets JMS préconfigurés créés par un administrateur afin d'être utilisés par les clients. Deux sortes :
  - destinations
  - connexions factories

Les outils d'administrations permettent de lier les fabriques de destinations et de connexions dans JNDI. Un client peut alors utiliser l'injection de ressource pour accéder à ces objets dans l'espace de nommage JNDI et établir une connexion à travers le provider JMS.



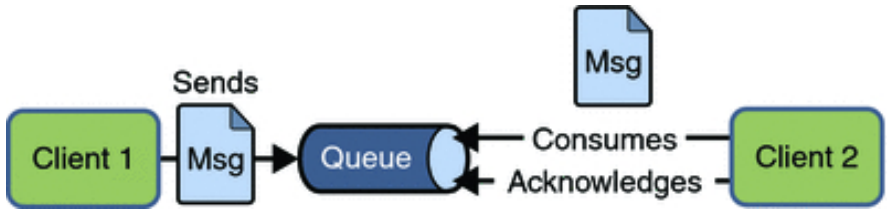
Avant l'arrivée de JMS, la plupart des produits fournissaient, soit l'approche *point-to-point*, soit l'appproche *publish/subscribe*.

JMS fournit un domaine différent pour chaque approche.

Un fournisseur JEE doit offrir ces deux domaines.

# Point-to-Point Messaging Domain

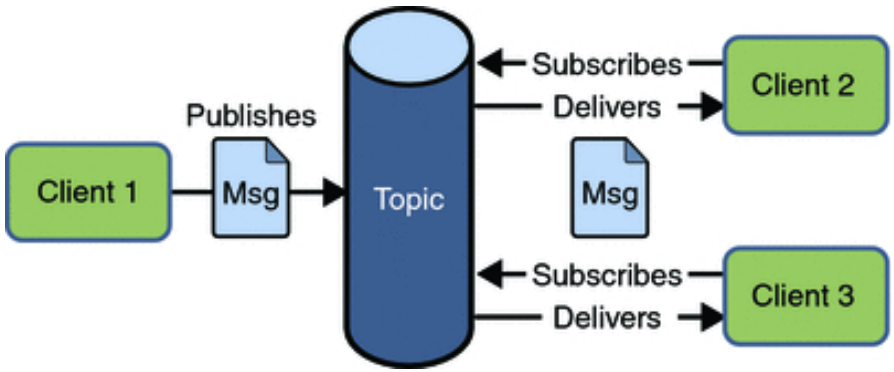
Une application point à point utilise les concepts de file de messages, d'émetteurs et de receveurs. Chaque message est adressé à une file donnée et chaque client extraie les messages depuis les files. Chaque file conserve les messages qui lui sont envoyés jusqu'à ce que le message soit consommé ou expire.




- Chaque message n'a qu'un seul consommateur.
- L'émetteur et le receveur du message ne sont pas synchronisés.
- Le receveur accuse réception du message.

# Publish/Subscribe Messaging Domain

Les clients envoient les messages à un sujet (*topic*). Les éditeurs (*publisher*) et les abonnés (*subscriber*) sont généralement anonymes et peuvent agir dynamiquement.
Le système s'occupe de l'acheminement des messages pour un sujet à éditeurs multiples aux multiples abonnés.
Les sujets ne conservent les messages que durant le temps nécessaires à leur distribution.



# Publish/Subscribe Messaging Domain

- Chaque message peut avoir plusieurs consommateurs.
- Editeurs et abonnés sont dépendants au niveau du temps. Un client ne peut consommer que les messages publiés après l'abonnement et doit rester actif.
- Une extension permet la flexibilité et la fiabilité des queues tout en permettant d'envoyer des messages à plusieurs destinataires : les abonnements durables.

JMS permet d'utiliser le même code pour envoyer et recevoir des messages en utilisant PTP ou Pub/Sub. Les destinations restent spécifiques au domaine mais le code peut être commun aux deux domaines.

JMS permet de consommer les messages de deux manières :

- de façon *synchrone* : un abonné ou un réceptionneur récupère explicitement le message en appelant la méthode `receive`. Celle ci peut bloquer jusqu'à l'arrivée d'un message ou jusqu'au timeout.

- de façon *asynchrone* : un client peut enregistrer un message `listener`. A l'arrivée d'un message, le fournisseur JMS délivre le message en appelant la méthode `onMessage` du `listener`.

# Sommaire

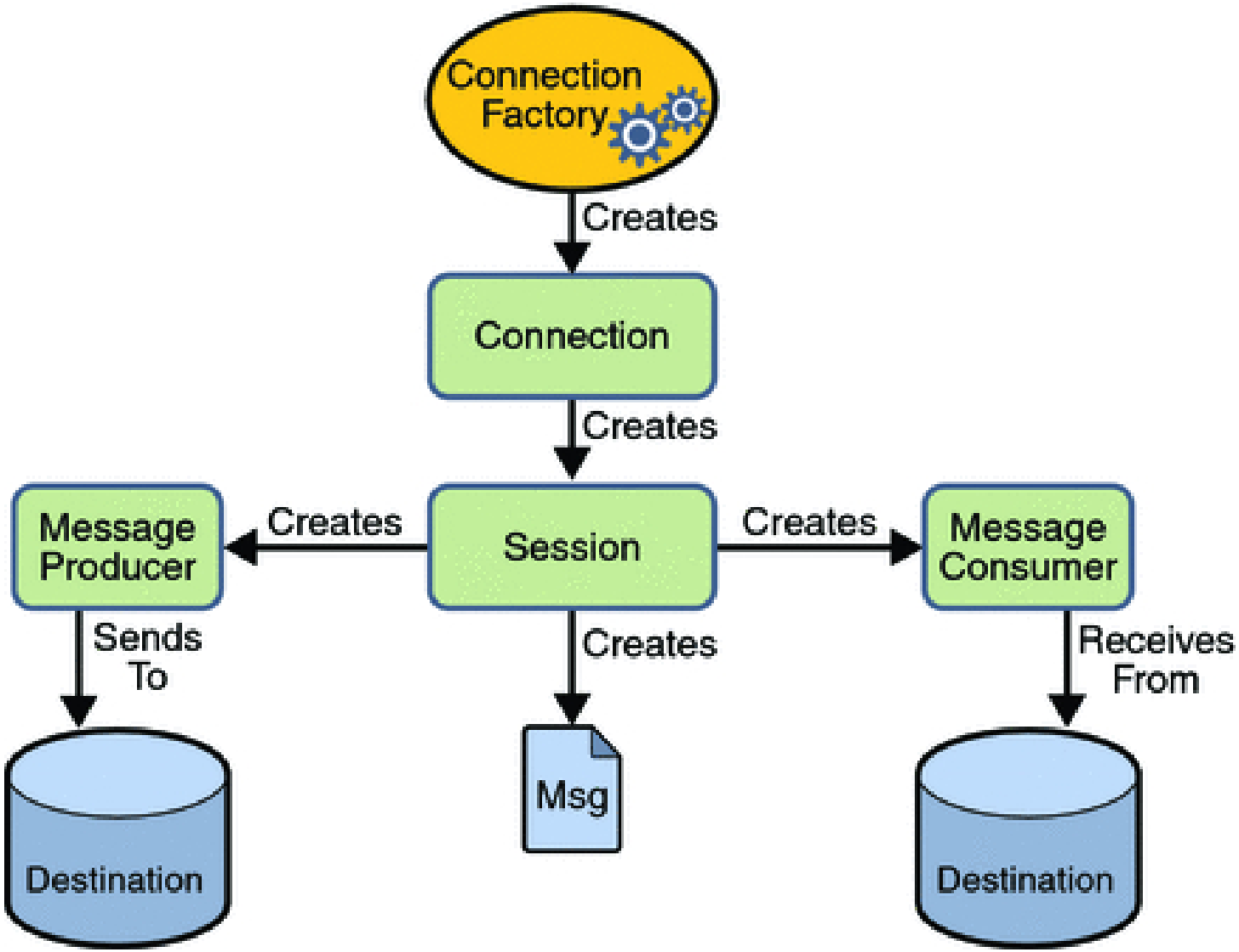
1 Introduction

2 Concepts de base JMS

3 Le modèle de programmation
- Une vue d'ensemble
- Les objets administrés
- Les connexions
- Les sessions
- Les producteurs de messages
- Les consommateurs de messages
- Les messages

4 Une application cliente

5 JMS dans une application JEE


Connection Factory
Creates
Connection
Creates
Message Producer
Creates
Session
Creates
Message Consumer
Sends To
Creates
Receives From
Msg
Destination
Destination

- Les fabriques de connexions et les destinations sont plus facilement gérables en les administrant plutôt qu'en les programmant. La technologie sous-jacente diffère d'une implémentation à l'autre et donc la façon de les administrer varie d'un fournisseur à l'autre.

- Les clients accèdent à ces objets au travers d'interfaces portables : ils n'ont pas à être modifiés d'une implémentation à l'autre. En général, l'administrateur lie ces objets dans un espace de nommage JNDI et les clients y accèdent en utilisant l'injection de ressource.

- Avec Glassfish, on pourra utiliser la commande `asadmin` ou la console d'administration pour créer ces fabriques sous forme de ressources.

# Fabriques de connexions

- C'est ce qu'utilise un client pour créer une connexion vers un fournisseur. Elle encapsule un ensemble de paramètres de configuration définis par l'administrateur. C'est une instance de l'interface `ConnectionFactory`, `QueueConnectionFactory` ou `TopicConnectionFactory`.
- A la création d'un client JMS, on injecte une ressource fabrique de connexions dans un objet `ConnectionFactory`. Le code suivant spécifie une ressource dont le nom JNDI est `jms/ConnectionFactory` et l'affecte à un objet `ConnectionFactory` :

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

- Dans une application JEE, les objets administrés JMS sont censés être placés dans le sous contexte de nommage `jms`.

# Les destinations

- Un client utilise un objet destination pour spécifier la cible du message qu'il produit et la source du message qu'il consomme. Pour le PTP, les destinations sont appelées `queues` et pour le pub/sub, `topics`.
- Pour créer une destination dans le serveur d'application, il faut créer une ressource destination JMS en lui assignant un nom JNDI.
- Dans le serveur d'application, chaque ressource destination fait référence à une destination physique. On peut les créer explicitement ou laisser le serveur créer la destination physique au besoin.
- Comme pour la fabrique de connexions, on injecte une ressource destination. Par contre, les destinations sont spécifiques à un domaine. Pour écrire du code pouvant être utilisé dans l'un ou l'autre domaine, il faut affecter la destination à un objet Destination.

# Les destinations

- Le code suivant spécifie deux ressources, une `queue` et un `topic`. Les noms des ressources sont projetés vers les destinations créées dans l'espace de nommage JNDI.

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

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

- On peut mixer les fabriques de connexions et les destinations : injecter une ressource `QueueConnectionFactory` et l'utiliser avec un topic ou injecter une ressource `TopicConnectionFactory` et l'utiliser avec une `queue`. Le comportement ne dépend que de la destination et pas de la fabrique de connexions.

Une connexion encapsule une connexion virtuelle au fournisseur JMS. Ca peut être une connexion TCP/IP entre le client et un démon du fournisseur. On l'utilise pour créer une ou plusieurs sessions.

Les connexions implémentent l'interface `Connection`. On utilise un objet `ConnectionFactory` pour créer une connexion :

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

Avant de terminer une application, il faut fermer toutes les connexions créées sinon le fournisseur risque de ne pas libérer les ressources associées. Cela a pour effet de fermer également ses sessions et les producteurs et consommateurs de messages : `connection.close();`

Avant de pouvoir consommer des messages, il faut appeler la méthode `start` sur la connexion. Pour temporairement stopper la délivrance des messages, on peut utiliser la méthode `stop`.

Une session est un contexte d'exécution à un seul flot pour produire et consommer des messages. On l'utilise pour créer des :

- producteurs de messages
- consommateurs de messages
- messages
- navigateurs de queues
- queues et topics temporaires

Elle sérialise l'exécution des message listeners.

Elle fournit un contexte transactionnel dans lequel regrouper un ensemble d'envois et de réceptions en une unité atomique.

Les sessions implémentent l'interface `Session`. On utilise un objet `Connection` pour créer une session :

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

Le premier paramètre signifie que la session n'est pas transactionnelle et le second que la session accuse réception automatiquement.

Pour créer une session transactionnelle, on utilise le code suivant :

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

Le premier paramètre signifie que la session est transactionnelle et le second que l'accusé de réception des messages n'est pas spécifié pour les sessions transactionnelles.

Un producteur de messages est un objet créé par une session et utilisé pour envoyer des messages à une destination. Il implémente l'interface `MessageProducer`.

On utilise une `Session` pour créer un `MessageProducer` pour une destination. L'exemple suivant crée un producteur pour un objet `Destination`, un objet `Queue`, ou un objet `Topic` :

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

On peut créer un producteur non identifié en donnant `null` comme paramètre. On ne spécifie la destination qu'à l'envoi de message.

Après avoir créé un producteur de message, on peut l'utiliser pour envoyer des messages avec la méthode `send` :

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

Il faut bien sur avoir créé le message au préalable.

Pour un producteur non identifié, on utilise la méthode surchargée `send` qui spécifie la destination en premier paramètre :

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

Un consommateur de messages est un objet créé par une session et utilisé pour recevoir des messages envoyés à une destination. Il implémente l'interface `MessageConsumer`.

Il permet à un client JMS de s'enregistrer à une destination auprès du fournisseur JMS. Celui ci gère la délivrance des messages d'une destination aux consomateurs qui y sont abonnés.

L'exemple suivant crée un consommateur pour un objet `Destination`, un objet `Queue`, ou un objet `Topic` :

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

La méthode `Session.createDurableSubscriber` permet de créer un abonné durable à un topic.

Dès sa création, un consommateur devient actif et peut recevoir des messages. La méthode `stop` permet de le rendre inactif. Les messages ne peuvent arriver qu'après avoir appelé la méthode `start` sur la connexion.

Pour recevoir un message de façon synchrone, on utilise la méthode `receive` :

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

Pour recevoir un message de façon asynchrone, on utilise un message listener.

# Message Listener

Un message listener est un objet qui agit comme un gestionnaire d'événements asynchrone. Il implémente l'interface `MessageListener` qui définit une seule méthode : `onMessage`. On définit dans celle ci les actions à prendre à l'arrivée d'un message.

Le message listener est enregistré auprès d'un `MessageConsumer` avec la méthode `setMessageListener`. Par exemple, la classe `Listener` implémentant l'interface `MessageListener`, on peut enregistrer le message listener comme suit :

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

Après avoir enregistré le message listener, on peut appeler la méthode start sur la connexion.

# Message Listener

Quand les messages commencent à être délivrés, le fournisseur JMS appelle automatiquement la méthode `onMessage`. Cette méthode a un paramètre de type `Message` que l'implémentation de la méthode peut caster dans le type voulu.

Un message listener n'est pas spécifique à un type de destination (queue ou topic) mais il attend en général un type de message bien précis.

La méthode `onMessage` doit rattraper toutes les exceptions. Elle ne doit lever aucune exception.

La session utilisée pour créer le consommateur de messages sérialise l'exécution de tous les message listener qui lui sont enregistrés. A tout moment, un seul est actif.

Un message-driven bean est en quelque sorte un message listener.

# Message Selectors

Si on a besoin de filtrer les messages reçus, on peut utiliser un sélecteur de messages : on spécifie les messages intéressants. Cela permet de laisser le travail de filtrage au fournisseur JMS plutôt qu'à l'application.

Un sélecteur de message est une chaîne de caractères qui contient une expression dont la syntaxe est un sous-ensemble de SQL92. L'exemple suivant sélectionne tout message qui a la propriété `NewsType` à `'Sport'` ou `'Opinion'` :

```
NewsType = 'Sports' OR NewsType = 'Opinion'
```

Les méthodes `createConsumer` et `createDurableSubscriber` permettent de spécifier un sélecteur de messages comme paramètre à la création du consommateur.

Le consommateur ne reçoit que les messages dont l'entête et les propriétés correspondent au sélectionneur. La sélection ne peut se faire sur le contenu.

Les messages JMS ont un format simple mais très flexible qui permet de créer des messages correspondant à des formats utilisés par une application non JMS.

Un message JMS possède trois parties :

- une entête
- des propriétés
- un corps

# Message headers

Une entête de message contient un certain nombre de champs prédéfinis qui contiennent des valeurs que les clients et les fournisseurs utilisent pour identifier et router les messages.

Par exemple, chaque message a un identifiant unique consultable dans le champ `JMSMessageID`. La valeur du champ `JMSDestination` représente la queue ou le topic à qui le message est envoyé. D'autres champs incluent un estampillage et un niveau de priorité.

Chaque champ possède son `getter` et son `setter`. Certains champs sont censés être positionnés par le client mais la plupart le sont automatiquement par la méthode `send` ou `publish`.

# Message headers

| 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 |

# Message properties

On peut créer et positionner des propriétés sur le message si on a besoin de valeurs supplémentaires aux champs de l'entête, par exemple pour être compatible avec un autre MOM ou pour utiliser un sélecteur de message.

JMS fournit des noms de propriétés prédéfinis qu'un fournisseur doit supporter.

# Message bodies

5 formats de corps de message : types de messages

| 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 serialized object in the Java programming language. |
| Message | Nothing. Composed of header fields and properties only, useful when a message body is not required. |

L'API fournit des méthodes pour créer des messages de chaque type. L'exemple suivant crée et envoie un `TextMessage` :

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

A la réception, le message est un objet générique de type `Message`. Il doit être casté au bon type. Un ou plusieurs getter peuvent être utilisés pour en extraire le contenu. Ici un exemple avec un `TextMessage` :

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

# Queue browser

On peut utiliser un objet `QueueBrowser` pour examiner les messages d'une queue. Les messages restent dans la queue jusqu'à ce que le consommateur les consomme. Un `QueueBrowser` permet de naviguer parmis les messages et de consulter l'entête de chaque message. Il se crée à partir de la session :

```
QueueBrowser browser = session.createBrowser(queue);
```

La méthode `createBrowser` permet de spécifier un sélecteur de messages en deuxième paramètre.

Impossible pour un topic.

# Sommaire

1 Introduction

2 Concepts de base JMS

3 Le modèle de programmation

4 Une application cliente
- Le producteur
- Le consommateur synchrone
- Le consommateur asynchrone

5 JMS dans une application JEE

```
public class Producer {
  @Resource(mappedName = "jms/ConnectionFactory")
  private static ConnectionFactory connectionFactory;
  @Resource(mappedName = "jms/Queue")
  private static Queue queue;
  @Resource(mappedName = "jms/Topic")
  private static Topic topic;

  public static void main(String[] args) {
    final int NUM_MSGS;
    Connection connection = null;
    if ((args.length < 1) || (args.length > 2)) {
      System.err.println(
        "Program takes one or two arguments: "
        + "<dest_type> [<number-of-messages>]");
      System.exit(1);
    }
```

```
String destType = args[0];
System.out.println("Destination type is " + destType);
if (!(destType.equals("queue") || destType.equals("topic"))) {
  System.err.println("Argument must be queue or topic");
  System.exit(1);
}
if (args.length == 2) {
  NUM_MSGS = (new Integer(args[1])).intValue();
} else {
  NUM_MSGS = 1;
}

Destination dest = null;

try {
  if (destType.equals("queue")) {
    dest = (Destination) queue;
  } else {
    dest = (Destination) topic;
  }
} catch (Exception e) {
  System.err.println("Error setting destination: "+e.toString());
  e.printStackTrace();
  System.exit(1);
}
```

```
    try {
      connection = connectionFactory.createConnection();

      Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
      MessageProducer producer = session.createProducer(dest);
      TextMessage message = session.createTextMessage();

      for (int i = 0; i < NUM_MSGS; i++) {
        message.setText("This is message " + (i + 1));
        System.out.println("Sending message: " + message.getText());
        producer.send(message);
      }

      /* Send a non-text control message indicating end of messages. */
      producer.send(session.createMessage());
    } catch (JMSException e) {
      System.err.println("Exception occurred: " + e.toString());
    } finally {
      if (connection != null) {
        try { connection.close(); } catch (JMSException e) {}
      }
    }
  }
}
```

```
public class SynchConsumer {
  @Resource(mappedName = "jms/ConnectionFactory")
  private static ConnectionFactory connectionFactory;
  @Resource(mappedName = "jms/Queue")
  private static Queue queue;
  @Resource(mappedName = "jms/Topic")
  private static Topic topic;

  public static void main(String[] args) {
    String destType = null;
    Connection connection = null;
    Session session = null;
    Destination dest = null;
    MessageConsumer consumer = null;
    TextMessage message = null;

    if (args.length != 1) {
      System.err.println("One argument: <dest_type>");
      System.exit(1);
    }

    destType = args[0];
    System.out.println("Destination type is " + destType);
```

```
if (!(destType.equals("queue") || destType.equals("topic"))) {
  System.err.println("Argument must be queue or topic");
  System.exit(1);
}

try {
  if (destType.equals("queue")) {
    dest = (Destination) queue;
  } else {
    dest = (Destination) topic;
  }
} catch (Exception e) {
  System.err.println("Error destination: "+e.toString());
  e.printStackTrace();
  System.exit(1);
}
```

```
    try {
      connection = connectionFactory.createConnection();
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      consumer = session.createConsumer(dest);

      connection.start();

      while (true) {

        Message m = consumer.receive(1);

        if (m != null) {
          if (m instanceof TextMessage) {
            message = (TextMessage) m;
            System.out.println("Reading message: "+ message.getText());
          } else {
            break;
          }
        }
      }
    } catch (JMSException e) {
      System.err.println("Exception occurred: " + e.toString());
    } finally {
      if(connection != null){try {connection.close();}catch (JMSException e) {}}
    }
  }
}
```

```
public class AsynchConsumer {
  @Resource(mappedName = "jms/ConnectionFactory")
  private static ConnectionFactory connectionFactory;
  @Resource(mappedName = "jms/Queue")
  private static Queue queue;
  @Resource(mappedName = "jms/Topic")
  private static Topic topic;

  public static void main(String[] args) {
    String destType = null;
    Connection connection = null;
    Session session = null;
    Destination dest = null;
    MessageConsumer consumer = null;
    TextListener listener = null;
    TextMessage message = null;
    InputStreamReader inputStreamReader = null;
    char answer = '\0';

    if (args.length != 1) {
      System.err.println("Program takes one argument: <dest_type>");
      System.exit(1);
    }

    destType = args[0];
    System.out.println("Destination type is " + destType);
```

```
if (!(destType.equals("queue") || destType.equals("topic"))) {
  System.err.println("Argument must be queue or topic");
  System.exit(1);
}

try {
  if (destType.equals("queue")) {
    dest = (Destination) queue;
  } else {
    dest = (Destination) topic;
  }
} catch (Exception e) {
  System.err.println("Error setting destination: " + e.toString());
  e.printStackTrace();
  System.exit(1);
}
```

```
    try {
      connection = connectionFactory.createConnection();
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      consumer = session.createConsumer(dest);

      listener = new TextListener();
      consumer.setMessageListener(listener);

      connection.start();

      System.out.println("To end program, type Q or q, " + "then <return>");
      inputStreamReader = new InputStreamReader(System.in);
      while (!((answer == 'q') || (answer == 'Q'))) {
        try {
          answer = (char) inputStreamReader.read();
        } catch (IOException e) {
          System.err.println("I/O exception: " + e.toString());
        }
      }
    } catch (JMSException e) {
      System.err.println("Exception occurred: " + e.toString());
    } finally {
      if (connection != null) {
        try { connection.close(); } catch (JMSException e) { }
      }
    }
  }
}
```

```java
public class TextListener implements MessageListener {
  /**
   * Casts the message to a TextMessage and displays its text.
   *
   * @param message the incoming message
   */

  public void onMessage(Message message) {
    TextMessage msg = null;

    try {
      if (message instanceof TextMessage) {
        msg = (TextMessage) message;
        System.out.println("Reading message: " + msg.getText());
      } else {
        System.err.println("Message is not a TextMessage");
      }
    } catch (JMSException e) {
      System.err.println("JMSException in onMessage(): " + e.toString());
    } catch (Throwable t) {
      System.err.println("Exception in onMessage():" + t.getMessage());
    }
  }
}
```

# Sommaire

1 Introduction

2 Concepts de base JMS

3 Le modèle de programmation

4 Une application cliente

5 JMS dans une application JEE
- L'annotation ressource
- Gestion des ressources

Dans une application cliente, on déclare la ressource JMS static :

```
@Resource(mappedName="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
@Resource(mappedName="jms/Queue")
private static Queue queue;
```

Mais quand on l'utilise dans un session bean, un message-driven bean, un composant web il ne faut pas déclarer la ressource static :

```
@Resource(mappedName="jms/ConnectionFactory")
private ConnectionFactory connectionFactory;
@Resource(mappedName="jms/Topic")
private Topic topic;
```

Si on la déclare statique, des erreurs se produiront.

Une application JEE qui doit produire ou recevoir des messages peut utiliser un session bean pour le faire.

Un appel bloquant à `receive` consomme des ressources du serveur : ne pas utiliser dans un bean. Il vaut mieux alors utiliser un mesage driven bean.

Les ressources JMS sont les connexions et les sessions. Il est important de les libérer quand on en a plus l'utilité.

Si on désire n'utiliser la ressource que pendant la durée de vie d'une méthode métier, on peut fermer la ressource dans un bloc finally dans le corps de la méthode.

Si on désire maintenir la ressource pour la durée de vie d'une instance d'un bean, il convient d'utiliser les méthodes `@PostConstruct` pour créer la ressource et `@PreDestroy` pour la fermer. S'il s'agit d'un stateful session bean, on doit fermer la ressource dans la méthode `@PrePassivate` et la mettre à null, et la recréer dans la méthode `@PostActivate`.