# NSY102

## Conception de logiciels Intranet : Patrons et Canevas.

Session de Juin 2018-durée : 2 heures

**Tous documents papiers autorisés**

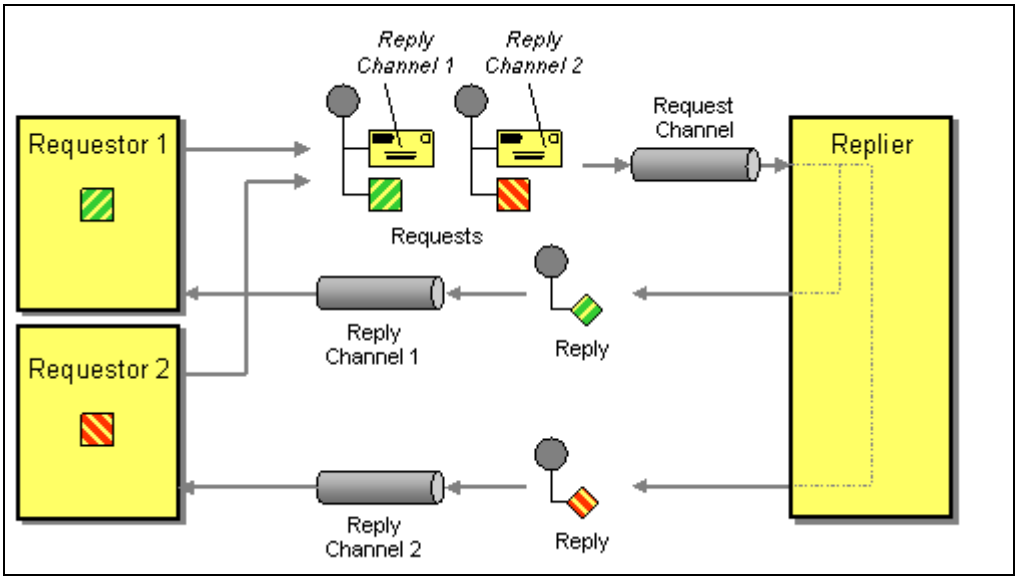
## Cnam / Paris-FOD Nationale

**Sommaire :**



## Question1: Patron *Return Address*

Le message envoyé contient entre autres une « Adresse de Retour » qui désigne de fait le récepteur du résultat.

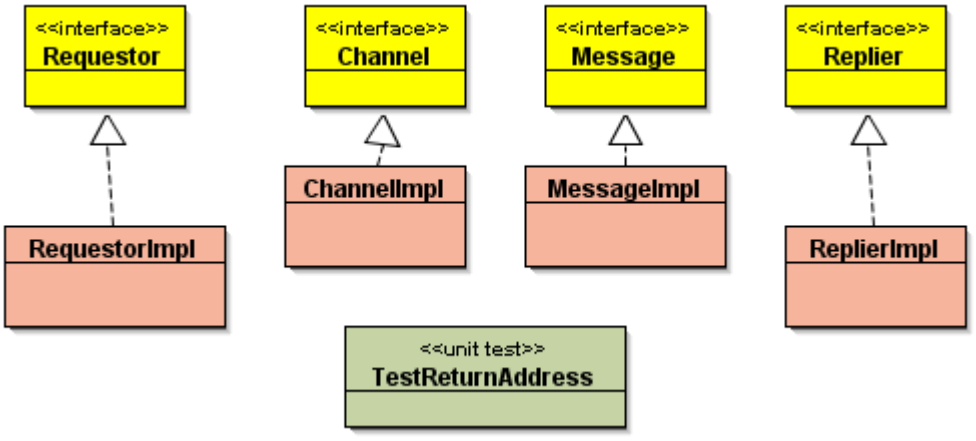



Source : http://www.enterpriseintegrationpatterns.com/ReturnAddress.html

Le message contient un canal de communication (*ReplyChannel*), sur lequel est envoyé le traitement effectué par le receveur (*Replier*). Cette encapsulation permet au receveur, d'« ignorer » l'émetteur (*Requestor*), soit sur le schéma, le demandeur du traitement à accomplir (Requestor1 pour Reply Channel 1 et Requestor2 pour Reply Channel 2).

Le message est transmis au receveur par le canal (*RequestChannel*). Ce message contient un « objet java » sérialisé ,, , le canal pour le retour et également un champ nommé *correlationId*

Ci dessous, une architecture de classes Java en notation UML/BlueJ mettant en oeuvre ce Patron,



- L'interface ***Channel,*** représente les canaux de communication

```
/** Canal de communication entre un émetteur et un récepteur.
 * Une file de messages de type LinkedBlockingQueue en interne
 * est utilisée.
 * @see java.util.concurrent.LinkedBlockingQueue
 */
public interface Channel{

  /** Envoi d'un message sur ce canal.
    * Appel non bloquant, la file interne a une capacité ad'hoc (illimitée).
    * @param message le message envoyé
    */
  public void send(Message message);

  /** Réception d'un message.
   * Le récepteur est bloqué si aucun message n'est disponible.
   * @param timeOut le délai de garde au delà duquel
   * une exception (InterruptedException) est levée.
   * @return le message reçu
   * @throws InterruptedException Lorsque le délai de garde est dépassé
   */
  public Message receive(long timeOut) throws InterruptedException;
}
```

- L'interface ***Requestor,*** est à implémenter par les émetteurs,

```
public interface Requestor {

  /** Envoi d'un message sur le canal RequestChannel.
   * (Le canal RequestChannel est affecté lors de la construction de l'objet)
   * @param message le message à transmettre
   */
  public void send(Message message) throws Exception;

  /** Réception d'un message.
   * La lecture est bloquante, une exception est levée si le délai de
   * garde est dépassée.
   */
  public Message receive(long timeout)throws InterruptedException;


}
```

- L'interface ***Replier,*** est à implémenter par les receveurs,
  - les receveurs ont un *thread* en interne, dont l'exécution permet :
    - d'attendre un message sur le canal en entrée,
    - d'envoyer le résultat dans un nouveau message sur le canal *ReplyChannel*

```
public interface Replier extends Runnable{

  /* Retourne le dernier message reçu.
   * @param msg le dernier message reçu ou null
   */
  public Message getTheLastMessageReceived() ;

}
```

- L'interface ***Message*** ci-dessous, contient les accès au contenu du message, ainsi qu'au canal pour la réponse et au champ de corrélation.

```
public interface Message{

  public Object getContent();
  public Channel getReplyChannel();
  public long getCorrelationId();

}
```

Une implémentation de cette interface ***Message*** est fournie :

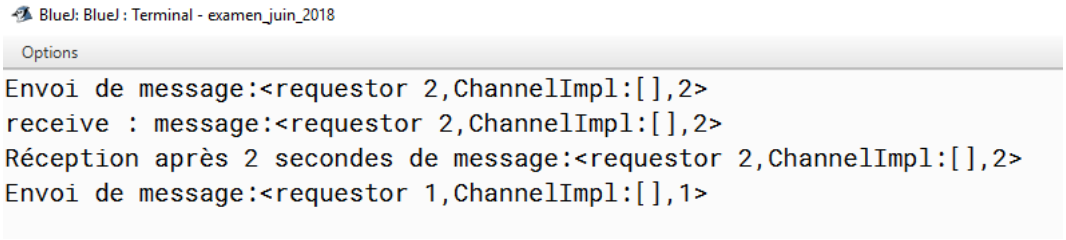
```
public class MessageImpl implements Message{
  private Object  content;
  private Channel replyChannel;
  private long    correlationId;

  public MessageImpl(Object content, Channel replyChannel, long correlationId){
    this.content       = content;
    this.replyChannel  = replyChannel;
    this.correlationId = correlationId;
  }

  public MessageImpl(Message message){
    this.content       = message.getContent();
    this.replyChannel  = message.getReplyChannel();
    this.correlationId = message.getCorrelationId();
  }

  public Channel getReplyChannel(){return replyChannel; }
  public Object getContent(){return content; }

  public String toString(){
    return "message:<" + content + "," + replyChannel + "," + correlationId + ">";
  }

  public long getCorrelationId(){return correlationId; }

  public boolean equals(Object obj){
    if(!(obj instanceof Message)) return false;
    Message msg = (Message) obj;
    return content.equals(msg.getContent()) &&
           replyChannel.equals(msg.getReplyChannel()) &&
           correlationId==msg.getCorrelationId();
  }
}
```

Ci-dessous la classe ***TestReturnAddress*** : une classe de tests, reflétant cette architecture qui est à lire **attentivement** …

```
public class TestReturnAddress extends junit.framework.TestCase{

 // selon http://www.enterpriseintegrationpatterns.com/ReturnAddress.html
   public void testReturnAddressPattern() throws Exception{
     // les canaux request, Reply1, Reply2
     Channel requestChannel = new ChannelImpl();
     Channel replyChannel_1 = new ChannelImpl();
     Channel replyChannel_2 = new ChannelImpl();

     // Requestor 1 et Requestor 2
     Requestor requestor_1 = new RequestorImpl(requestChannel,replyChannel_1);
     Requestor requestor_2 = new RequestorImpl(requestChannel,replyChannel_2);

      System.out.println("Envoi de " + message2);
     // Replier avec un délai de garde de 10 sec
     Replier replier = new ReplierImpl(requestChannel, 10000);
```

```
        // création de deux messages
        Message message1 = new MessageImpl("requestor 1", replyChannel_1, 1L);
        Message message2 = new MessageImpl("requestor 2", replyChannel_2, 2L);

        requestor_2.send(message2);
        Thread.sleep(1000); // attente de une seconde
        Message received = requestor_2.receive(2000);
         System.out.println("Réception après 2 secondes de " + received);
        assertEquals("requestor 2", received.getContent().toString());

        requestor_1.send(message1);
        Thread.sleep(1000); // attente de une seconde
        received = requestor_1.receive(2000);
         System.out.println("Réception après 3 secondes de " + received);
        assertEquals("requestor 1", received.getContent().toString());


    }
}
```

La trace obtenue sur la console




**Question)**

- Ecrivez les 3 implémentations issues respectivement des interfaces **Requestor, Channel, Replier**
    - **RequestorImpl, ChannelImpl, ReplierImpl**

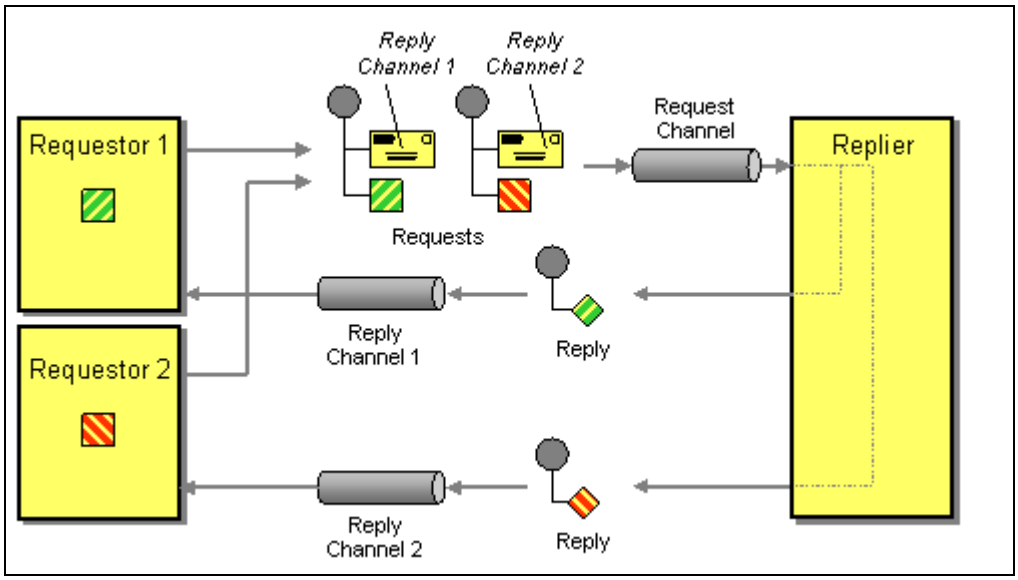
Notez que :

Pour la classe **RequestorImpl (implements Requestor)** l'appel de la méthode ***receive*** est bloquante (jusqu'à « l'arrivée » d'un message).

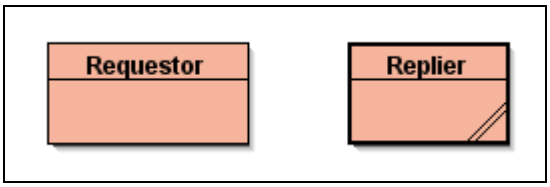
Pour la classe **ChannelImpl (implements Channel)** une file de messages synchronisée dont la taille peut croître dynamiquement est utilisée en interne, cf. par exemple LinkedBlockingQueue.

Pour la classe **ReplierImpl (implements Replier)** un « thread » interne permet de réceptionner un message et de le retourner tel quel, sans traitement ici, l'itération qui se poursuit tant que le délai garde n'est pas dépassé (10 secondes dans la classe de test ci-dessus). Au délai de garde échu un affichage d'un message est effectué et le récepteur est de nouveau prêt à recevoir.

## Question2 : Patron *ReturnAddress* avec JMS




**Question2)** Proposez le même patron en utilisant l'API JMS, JAVA Message Service, les files RequestChannel, ReplyChannel_1 et ReplyChannel_2 ont été créées par l'administrateur JMS.




Complétez les classes ***Requestor*** et ***Replier*** ci-dessous, en proposant respectivement les méthodes de classe *newRequestor* et *newReplier*

### La classe *Requestor* à compléter :

```
public class Requestor implements MessageListener {
  private static Context context;

        // à compléter sur votre copie
```

```
 public static void main(String[] args) throws Exception{

  Hashtable<String,String> properties = new Hashtable<String,String>();
```

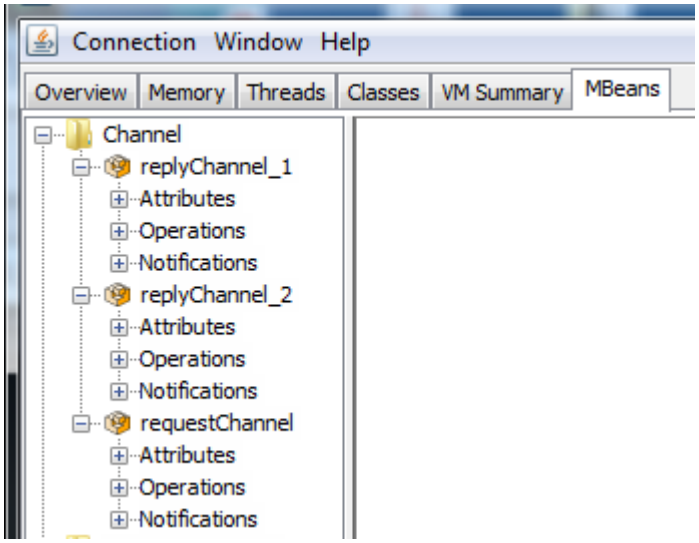
```
  properties.put(Context.INITIAL_CONTEXT_FACTORY, "org.exolab.jms.jndi.InitialContextFactory");
  properties.put(Context.PROVIDER_URL, "tcp://localhost:3035/");

  Requestor.context = new InitialContext(properties);

  // look up the ConnectionFactory
  ConnectionFactory factory = (ConnectionFactory) context.lookup("ConnectionFactory");

  // look up the Destination
  Destination requestChannel = (Destination) context.lookup("RequestChannel");

  // create the connection
  Connection connection = factory.createConnection();

  // args[0] le nom de la file : ReplyChannel_1 ou ReplyChannel_2
  Requestor requestor = Requestor.newRequestor(connection, requestChannel, args[0]);
  // args[1] le message
  requestor.send(args[1]); // le message
}
```

### La classe *Replier* à compléter

```
      public class Replier implements MessageListener {

      private Session session;
      private static Context context;public static void main(String[] args) throws
Exception{
    // create the JNDI initial context
    // nsy102 lignes extraites de http://openjms.sourceforge.net/usersguide/using.html
    Hashtable<String,String> properties = new Hashtable<String,String>();
    properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.exolab.jms.jndi.InitialContextFactory");
    properties.put(Context.PROVIDER_URL, "tcp://localhost:3035/");

    Replier.context = new InitialContext(properties);

    // look up the ConnectionFactory
    ConnectionFactory factory = (ConnectionFactory) context.lookup("ConnectionFactory");

      // create the connection
    Connection connection = factory.createConnection();

        Replier replier = Replier.newReplier(connection, "RequestChannel");

        Thread.sleep(Long.MAX_VALUE);
      }
}
```

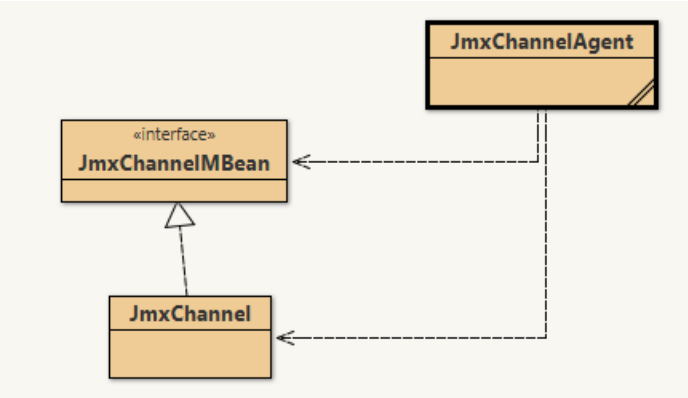
## Question3 : Patron *Return Address* avec JMX

**Question3)** Proposez une implémentation java du « Channel » de la question1 permettant avec la technologie JMX d'être notifié à chaque réception, émission et exception de messages sur un canal.




Une copie d'écran avec l'outil prédéfini jconsole,
ici les 3 canaux de l'exemple de la question1,
*replyChannel_1*, *replyChannel_2*, *requestChannel*

Proposez une interface *JmxChannelMBean* et une implémentation d'un canal *JmxChannel*, compatibles avec la technologie JMX. L'agent JMX qui se charge d'enregistrer les différents canaux auprès de la plateforme n'est pas demandé.




**javax.jms**
**Interface Connection**

**All Known Subinterfaces:**
QueueConnection, TopicConnection, XAQueueConnection, XATopicConnection

---

public interface **Connection**

A `Connection` object is a client's active connection to its JMS provider. It typically allocates provider resources outside the Java virtual machine (JVM).

Connections support concurrent use.

**See Also:**
`ConnectionFactory`, `QueueConnection`, `TopicConnection`

---

| Method Summary | |
|---|---|
| void | **start**()<br>Starts (or restarts) a connection's delivery of incoming messages. |

**javax.jms**
**Interface Destination**

**All Known Subinterfaces:**
Queue, TemporaryQueue, TemporaryTopic, Topic

---

public interface **Destination**

---

**javax.jms**
**Interface Message**

**All Known Subinterfaces:**
…, TextMessage

---

public interface **Message**

The `Message` interface is the root interface of all JMS messages. It defines the message header and the `acknowledge` method used for all messages.

| Method Summary | |
|---|---|
| `java.lang.String` | **`getJMSCorrelationID`**`()`<br>Gets the correlation ID for the message. |
| `java.lang.String` | **`getJMSMessageID`**`()`<br>Gets the message ID. |
| `Destination` | **`getJMSReplyTo`**`()`<br>Gets the `Destination` object to which a reply to this message should be sent. |
| `void` | **`setJMSReplyTo`**`(Destination replyTo)`<br>Sets the `Destination` object to which a reply to this message should be sent. |

**javax.jms**
**Interface MessageConsumer**

**All Known Subinterfaces:**
QueueReceiver, TopicSubscriber

---

public interface **MessageConsumer**

A client uses a `MessageConsumer` object to receive messages from a destination. A `MessageConsumer` object is created by passing a `Destination` object to a message-consumer creation method supplied by a session.

---

| Method Summary | |
|---|---|
| `Message` | **`receive`**`()`<br>Receives the next message produced for this message consumer. |
| `void` | **`setMessageListener`**`(MessageListener listener)`<br>Sets the message consumer's `MessageListener`. |

**javax.jms**
**Interface MessageListener**

---

public interface **MessageListener**

---

| Method Summary | |
|---|---|
| `void` | **`onMessage`**`(Message message)`<br>Passes a message to the listener. |

**javax.jms**
**Interface MessageProducer**

**All Known Subinterfaces:**
QueueSender, TopicPublisher

---

public interface **MessageProducer**

A client uses a `MessageProducer` object to send messages to a destination. A `MessageProducer` object is created by passing a `Destination` object to a message-producer creation method supplied by a session.

| Method Summary | |
|---|---|
| void | **send**(Message message)<br>Sends a message to the queue. |

**javax.jms**
**Interface TextMessage**

**All Superinterfaces:**
Message

---

public interface **TextMessage**
extends Message

A `TextMessage` object is used to send a message containing a `java.lang.String`. It inherits from the `Message` interface and adds a text message body.

**See Also:**
`Session.createTextMessage()`, `Session.createTextMessage(String)`, `BytesMessage`, `MapMessage`, `Message`, `ObjectMessage`, `StreamMessage`, `String`

---

| Method Summary | |
|---|---|
| java.lang.String | **getText**()<br>Gets the string containing this message's data. |
| void | **setText**(java.lang.String string)<br>Sets the string containing this message's data. |

**java.util.concurrent**
**Class LinkedBlockingQueue<E>**

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractQueue<E>
      **java.util.concurrent.LinkedBlockingQueue<E>**

**Type Parameters:**
`E` - the type of elements held in this collection
**All Implemented Interfaces:**
Serializable, Iterable<E>, Collection<E>, BlockingQueue<E>, Queue<E>

---

```
public class LinkedBlockingQueue<E>
extends AbstractQueue<E>
implements BlockingQueue<E>, Serializable
```

An optionally-bounded blocking queue based on linked nodes. This queue orders elements FIFO (first-in-first-out). The *head* of the queue is that element that has been on the queue the longest time. The *tail* of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue. Linked queues typically have higher throughput than array-based queues but less predictable performance in most concurrent applications.

The optional capacity bound constructor argument serves as a way to prevent excessive queue expansion. The capacity, if unspecified, is equal to `Integer.MAX_VALUE`. Linked nodes are dynamically created upon each insertion unless this would bring the queue above capacity.

This class and its iterator implement all of the *optional* methods of the `Collection` and `Iterator` interfaces.

This class is a member of the Java Collections Framework.

**Since:**
1.5
**See Also:**
Serialized Form

---

| Constructor Summary |
|---|
| **LinkedBlockingQueue**() |

| | |
|---|---|
| | Creates a `LinkedBlockingQueue` with a capacity of `Integer.MAX_VALUE`. |
| **LinkedBlockingQueue** (Collection<? extends E> c)<br>Creates a `LinkedBlockingQueue` with a capacity of `Integer.MAX_VALUE`, initially containing the elements of the given collection, added in traversal order of the collection's iterator. | |
| **LinkedBlockingQueue** (int capacity)<br>Creates a `LinkedBlockingQueue` with the given (fixed) capacity. | |

| Method Summary | |
|---|---|
| void | **clear** ()<br>Atomically removes all of the elements from this queue. |
| int | **drainTo** (Collection<? super E> c)<br>Removes all available elements from this queue and adds them to the given collection. |
| int | **drainTo** (Collection<? super E> c, int maxElements)<br>Removes at most the given number of available elements from this queue and adds them to the given collection. |
| Iterator<E> | **iterator** ()<br>Returns an iterator over the elements in this queue in proper sequence. |
| boolean | **offer** (E e)<br>Inserts the specified element at the tail of this queue if it is possible to do so immediately without exceeding the queue's capacity, returning `true` upon success and `false` if this queue is full. |
| boolean | **offer** (E e, long timeout, TimeUnit unit)<br>Inserts the specified element at the tail of this queue, waiting if necessary up to the specified wait time for space to become available. |
| E | **peek** ()<br>Retrieves, but does not remove, the head of this queue, or returns `null` if this queue is empty. |
| E | **poll** ()<br>Retrieves and removes the head of this queue, or returns `null` if this queue is empty. |
| E | **poll** (long timeout, TimeUnit unit)<br>Retrieves and removes the head of this queue, waiting up to the specified wait time if necessary for an element to become available. |
| void | **put** (E e)<br>Inserts the specified element at the tail of this queue, waiting if necessary for space to become available. |
| int | **remainingCapacity** ()<br>Returns the number of additional elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking. |
| boolean | **remove** (Object o)<br>Removes a single instance of the specified element from this queue, if it is present. |
| int | **size** ()<br>Returns the number of elements in this queue. |
| E | **take** ()<br>Retrieves and removes the head of this queue, waiting if necessary until an element becomes available. |
| Object[] | **toArray** ()<br>Returns an array containing all of the elements in this queue, in proper sequence. |
| <T> T[] | **toArray** (T[] a)<br>Returns an array containing all of the elements in this queue, in proper sequence; the runtime type of the returned array is that of the specified array. |
| String | **toString** ()<br>Returns a string representation of this collection. |

**javax.management**
**Interface NotificationBroadcaster**

**All Known Subinterfaces:**
ModelMBean, ModelMBeanNotificationBroadcaster, NotificationEmitter
**All Known Implementing Classes:**
CounterMonitor, GaugeMonitor, JMXConnectorServer, MBeanServerDelegate, Monitor, NotificationBroadcasterSupport, RelationService, RequiredModelMBean, RMIConnectorServer, StandardEmitterMBean, StringMonitor, Timer

---

`public interface` **`NotificationBroadcaster`**

Interface implemented by an MBean that emits Notifications. It allows a listener to be registered with the MBean as a notification listener.

---

| Method Summary |
|---|

| | |
|---|---|
| void | **addNotificationListener**(NotificationListener listener, NotificationFilter filter, Object handback)<br>Adds a listener to this MBean. |
| MBeanNotificationInfo[] | **getNotificationInfo**()<br>Returns an array indicating, for each notification this MBean may send, the name of the Java class of the notification and the notification type. |
| void | **removeNotificationListener**(NotificationListener listener)<br>Removes a listener from this MBean. |

**javax.management**
**class NotificationBroadcasterSupport**

java.lang.Object
  └ **javax.management.NotificationBroadcasterSupport**

**All Implemented Interfaces:**
NotificationEmitter
**Direct Known Subclasses:**
JMXConnectorServer, Monitor, RelationService, Timer

---

**Most common way to construct:**

```
NotificationBroadcasterSupport broadcaster = new NotificationBroadcasterSupport();
```

*Based on 50 examples*

---

```
public class NotificationBroadcasterSupport
extends Object
implements NotificationEmitter
```

| Constructor Summary |
|---|
| **NotificationBroadcasterSupport**()<br>Constructs a NotificationBroadcasterSupport where each listener is invoked by the thread sending the notification. |
| **NotificationBroadcasterSupport**(Executor executor)<br>Constructs a NotificationBroadcasterSupport where each listener is invoked using the given java.util.concurrent.Executor. |
| **NotificationBroadcasterSupport**(Executor executor, MBeanNotificationInfo[] info)<br>Constructs a NotificationBroadcasterSupport with information about the notifications that may be sent, and where each listener is invoked using the given java.util.concurrent.Executor. |
| **NotificationBroadcasterSupport**(MBeanNotificationInfo[] info)<br>Constructs a NotificationBroadcasterSupport with information about the notifications that may be sent. |

| Method Summary | |
|---|---|
| void | **sendNotification**(Notification notification)<br>Sends a notification. |

## javax.management
**class Notification**

java.lang.Object
  └ java.util.EventObject
    └ **javax.management.Notification**

**All Implemented Interfaces:**
Serializable
**Direct Known Subclasses:**
AttributeChangeNotification, JMXConnectionNotification, MBeanServerNotification, MonitorNotification, RelationNotification, TimerNotification

---

```
public class Notification
extends EventObject
```

| Field Summary | |
|---|---|
| protected Object | **source**<br>This field hides the java.util.EventObject.source field in the parent class to make it non-transient and therefore part of the serialized form. |

| Fields inherited from class java.util.EventObject |
| --- |
| source |

| Constructor Summary |
| --- |
| **Notification**(String type, Object source, long sequenceNumber)<br>Creates a Notification object. |
| **Notification**(String type, Object source, long sequenceNumber, long timeStamp)<br>Creates a Notification object. |
| **Notification**(String type, Object source, long sequenceNumber, long timeStamp, String message)<br>Creates a Notification object. |
| **Notification**(String type, Object source, long sequenceNumber, String message)<br>Creates a Notification object. |