NSY102

Conception de logiciels Intranet : Patrons et Canevas.

Session de Juin 2011-durée : 2 heures

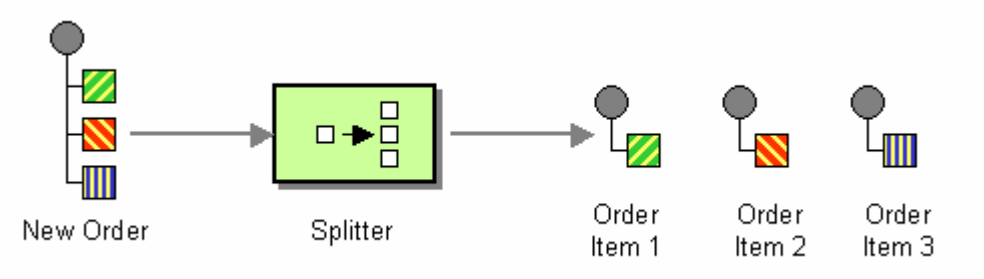
Tous documents papiers autorisés

Cnam / Paris-HTO & FOD

Sommaire :

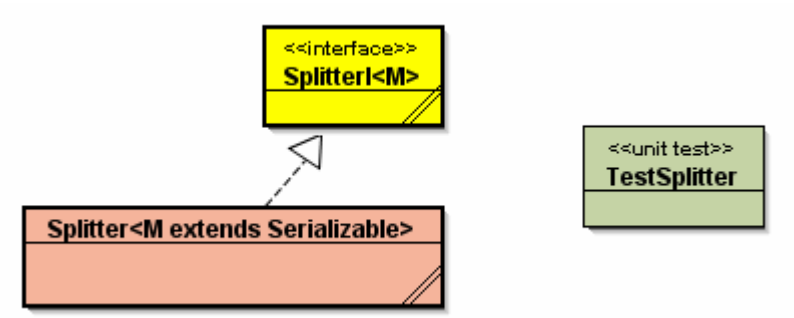
Question1 (6 points) : Patron *Splitter*

Ce patron permet de sérialiser un message composé. Le patron *Splitter* fournit en séquence le message initial, à destination de un ou plusieurs receveurs. Il garantit que tous les messages ont bien été lus, et qu'un nouveau message composé ne peut être envoyé sans que tous les messages sérialisés du précédent envoi n'aient été lus.



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

Ci dessous, une architecture de classes Java reflétant ce Patron, en notation Bluej/UML



Cette architecture est constituée de l'interface ***SplitterI*** d'une implémentation ***Splitter***, *ce qui est demandé,* et d'une classe de tests unitaires ***TestSplitter*** fournie, à lire attentivement.

L'interface ***SplitterI<M>*** ci-dessous**,** dans laquelle ***M*** représente le type des messages, propose les opérations :

- d'un envoi d'une table de messages (`send`). La table de messages représente le message composé, chaque élément du tableau est l'un des messages du message composé,
- de lecture d'un message (`get`),
- d'une réinitialisation de la table des messages (`clear`),
- d'obtention de la liste des indices des messages non lus (`unreadMessages`).

```
/** Le Patron Splitter sérialise le message envoyé.
 * @param <M> le type des messages
 */
public interface SplitterI<M> {

  /** Envoi d'une table de messages.
   * @param messages la table des messages
   * @throws Exception si la table est null,
   *                   ou si depuis le dernier envoi tous les messages n'ont pas été lus.
   */
  void send(M... messages) throws Exception;

  /** Lecture d'un message. Plusieurs lectures du même message sont possibles.
   * @param index l'indice du message
   * @throws Exception si le message n'existe pas, ou si cet index est erroné
   */
  M get(int index) throws Exception;

  /** Efface la dernière table des messages envoyée (exception ou non).
   * Une exception est levée si tous les messages n'ont pas été lus.
   * @throws Exception si depuis le dernier envoi tous les éléments n'ont pas été lus.
   */
  void clear() throws Exception;

  /** Obtention d'un ensemble contenant les indices des messages non lus depuis le dernier envoi.
   * @return un ensemble d'indices
   * @throws Exception si aucun envoi n'a eu lieu.
   */
  Set<Integer> unreadMessages() throws Exception;
}
```

Ci-dessous un extrait de la classe de tests unitaires **SplitterTest**, ces lignes sont à <u>étudier en détail</u> avant de répondre à cette question

```
public void testScenario1() throws Exception{
  final SplitterI<String> s = new Splitter<String>();
  s.send("zero","un","deux","trois","quatre");     // envoi d'une table de 5 'String'
  String msg = s.get(2);                          // lecture de l'élément d'indice 2
  assertEquals("deux",msg);

  Set<Integer> set = s.unreadMessages();          // obtention des éléments non lus
  assertTrue(set.contains(0));assertTrue(set.contains(1));
  assertTrue(set.contains(3));assertTrue(set.contains(4));
  assertEquals(4,set.size());                              // soit 4 éléments non lus

  assertEquals("quatre",s.get(4));                           // une lecture de plus
  assertEquals(3,s.unreadMessages().size());               // soit 3 éléments non lus
  try{
    s.send("test"); // nouvel envoi alors qu'il reste des éléments à lire : exception
    fail("une exception est attendue ici ...");
  }catch(Exception e){}

  for(int i : s.unreadMessages()){     // pour tous les messages non lus : les lire …
    String v = s.get(i);
  }

  String v = s.get(2);                        // une nouvelle lecture est possible
  assertEquals(0,s.unreadMessages().size());

  s.send("test0","test1");                       // nouvel envoi de deux éléments
  assertEquals(2,s.unreadMessages().size());
  try{
    s.clear();// clear() alors qu'il reste des éléments à lire : exception
    fail("une exception est attendue ici ...");
  }catch(Exception e){}
}
```

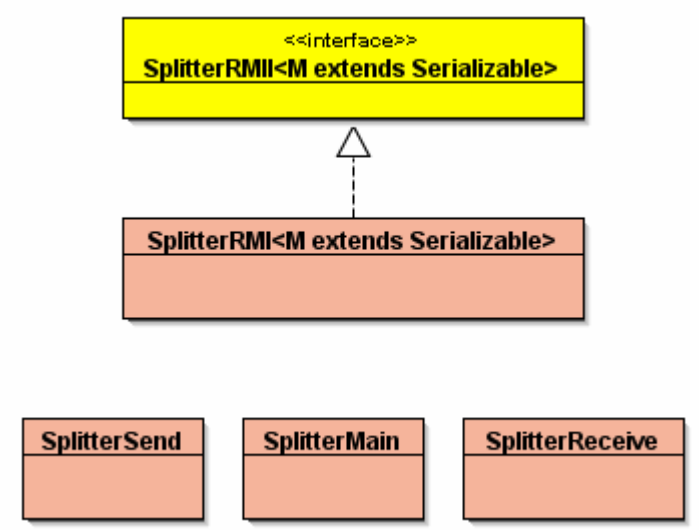
Question)
Ecrivez une implémentation **complète de la classe *Splitter***. L'implémentation demandée autorise les accès concurrents, en effet plusieurs clients (plusieurs Thread) pourraient accéder aux méthodes de l'instance simultanément. Certaines documentations d'interfaces ou de classes sont en annexe.

Question2 (8 points) : Patron *Splitter* en intranet / RMI

Question2-1) Proposez **un schéma d'une architecture matérielle** utilisant le patron *Splitter* en intranet avec la technologie RMI. Les émetteurs, les receveurs et le patron Splitter sont installés sur des machines différentes reliées entre-elles, dont les noms sont respectivement **splittersend.cnam.fr**, **splitterreceive.cnam.fr** et **splittermain.cnam.fr**.

Précisez quels sont les services, quels sont les clients et sur quelle(s) machine trouve-t-on l'(es) annuaire(s).

Question2-suite)
Soit ci-dessous, une architecture possible de classes de ce patron en version intranet/rmi :



L'interface ***SplitterRMII*** contient les mêmes noms de méthodes avec les mêmes fonctionnalités que l'interface ***SplitterI*** de la question 1. Les descriptions de certaines classes ou interfaces extraites des paquetages java.rmi, java.rmi.server et java.rmi.registry sont en annexe.

L'**interface SplitterRMII** a été adaptée pour RMI, par :

- l'ajout de la clause *throws java.rmi.**RemoteException*** pour chaque méthode,
- l'héritage au sens des interfaces de java.rmi.***Remote.***

```
import question1.SplitterI;
import …

public interface SplitterRMII<M extends Serializable> extends  Remote{

  void send(M... messages) throws RemoteException, Exception;

  M get(int index) throws RemoteException, Exception;

  void clear() throws RemoteException, Exception;
```

```
  Set<Integer> unreadMessages() throws RemoteException, Exception;

}
```

L'application java associée à la classe ***SplittertMain***, ci-dessous, est installée sur la machine nommée **splittermain.cnam.fr**. L'annuaire rmiregistry est présent.

```
import question1.Splitter;
import ……

public class SplitterMain{

 public static void main(String[] args) throws Exception {
    System.setSecurityManager(new RMISecurityManager());

    Remote splitter_stub = new SplitterRMI<Serializable>(new Splitter<Serializable>());
    Naming.rebind("splitter",splitter_stub); // l'annuaire par défaut c.f. rmiregistry

    System.out.println("splittermain.cnam.fr, service \"splitter\" installé");

  }
}
```

Question2-2) Proposez une implémentation **complète** de la classe ***SplitterRMI*.**

Question2-3) Ecrivez une implémentation de la classe SplitterSend, qui se contente d'envoyer une table composée de 5 éléments vers l'objet distant nommé **`"splitter"`** et d'attendre que ces éléments aient été lus avant de se terminer. Cette application est installée sur la machine **splittersend.cnam.fr.**

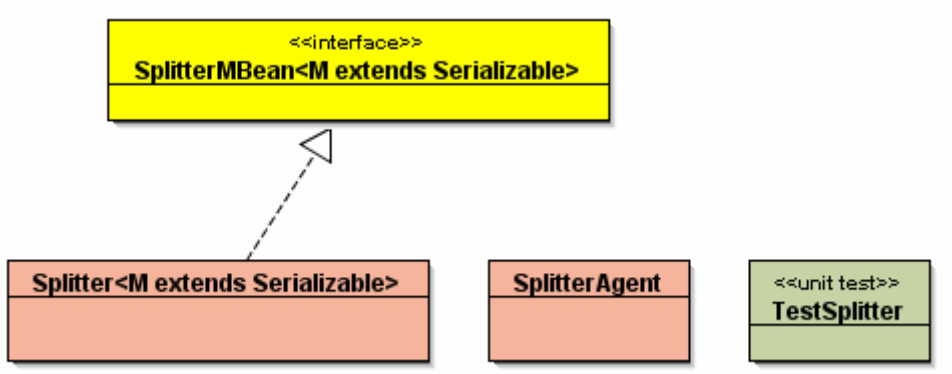
Question2-4) Ecrivez une implémentation de la classe SplitterReceive qui affiche sur la console tous les messages émis quels qu'il soient, cette application java ne se termine jamais. SplitterReceive est installée sur la machine **splittereceive.cnam.fr.**

Note : Ces deux applications (SplitterSend et SplitterReceive) installées sur deux machines distinctes s'exécutent dans n'importe quel ordre chronologique.

Question3 (6 points) : Le Patron *Splitter* / JMX

Le patron ***Splitter*** est maintenant supervisé avec la technologie JMX. A la question1, plusieurs exceptions sont levées en fonction du contexte. Nous souhaitons être prévenus, notifiés de toute exception levée par une implémentation du Splitter de la question1.

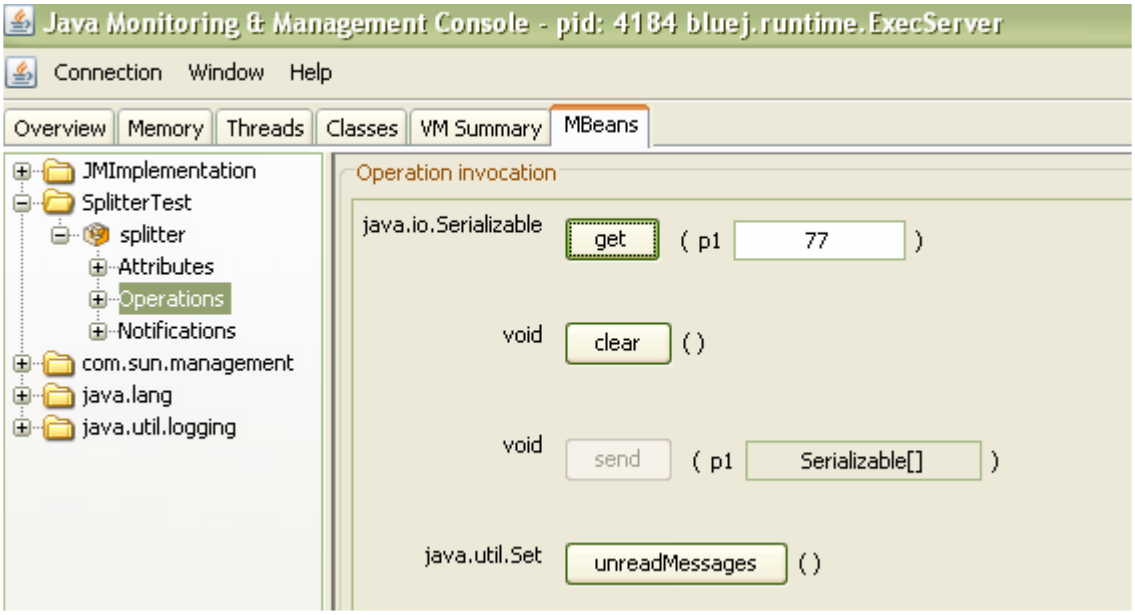
Ci-dessous, une architecture possible en utilisant l'implémentation issue de la question1



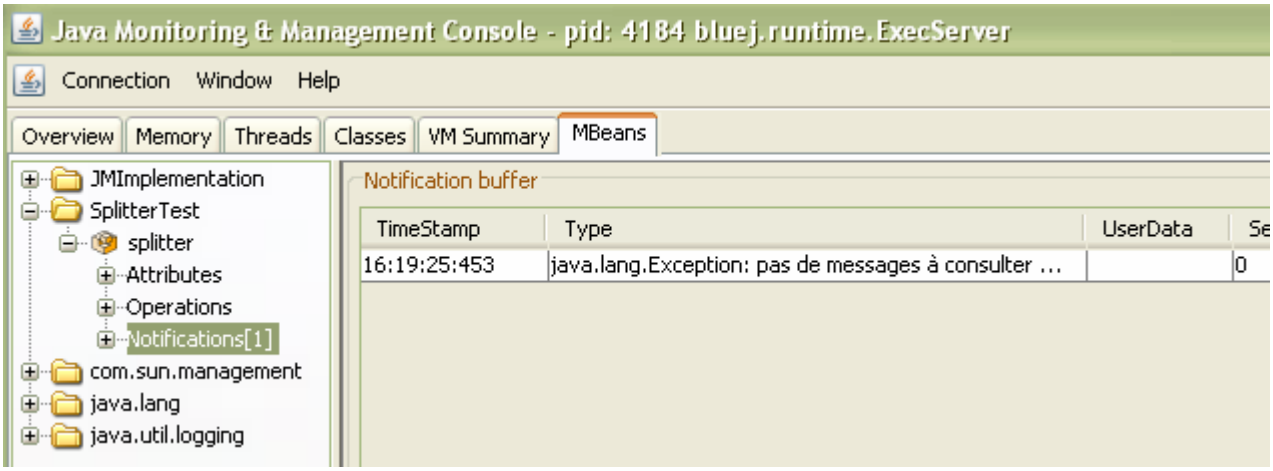
Un extrait de la classe de test ***TestSplitter*** .

```
static{
  new Thread(new Runnable(){
    public void run(){
      try{
        MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
        ObjectName beanName = new ObjectName( "SplitterTest:name=splitter" );

        SplitterMBean<String> mbean = null;
        mbean = new Splitter<String>(new question1.Splitter<String>());
        mbs.registerMBean(mbean, beanName);
      }catch(Exception e){
      }
    }}).start();
}

  public void testScenario1() throws Exception{
    Thread.sleep(1000*60); // une pause, le temps d'exécuter jconsole …
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
    ObjectName beanName = new ObjectName( "SplitterTest:name=splitter" );
    final SplitterMBean<String> s = null;
    s = (SplitterMBean<String>) MBeanServerInvocationHandler.newProxyInstance(
                                        mbs,
                                        beanName,
                                        SplitterMBean.class,
                                        true);
     // tests identiques aux tests de unitaires de la question1
……
  }
```

Ci-dessous : une copie d'écran de l'outil jconsole connecté au MBeanServer sur lequel le MBean nommé SplitterTest est installé



Après avoir souscrit et tenté d'accéder au 77ème élément d'un message qui n'en compte que 5, un deuxième client jconsole est notifié de cette exception.



Question3-1) Proposez l'interface *SplitterMBean* **et une implémentation complète** de la classe ***Splitter*.**

Question3-2) Une application distante utilise le connecteur RMI, déjà installé par l'agent. Cette application, un client RMI, souscrit aux notifications comme l'outil jconsole et se contente d'afficher le type et le nombre transmis par RecipientList lors de la notification.

Proposez une implémentation complète de la classe ***ClientRMI*.**

Certaines interfaces se trouvent en annexe.

------- fin ---------

Une idée de la solution sera en ligne dans quelques jours à cette URL :
http://jfod.cnam.fr/NSY102/annales/2011_juin/

Annexes

java.util
interface List<E> ***partielle***

All Superinterfaces:
Collection<E>, Iterable<E>
All Known Implementing Classes:
LinkedList, … ArrayList, Vector

---

public interface **List<E>**
extends Collection<E>

Method Summary	
boolean	**add**(E e) Appends the specified element to the end of this list (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(Object o) Returns true if this list contains the specified element.
boolean	**isEmpty**() Returns true if this list contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
boolean	**remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
int	**size**() Returns the number of elements in this list.

---

java.util
Interface Map<K,V>

Type Parameters:
K - the type of keys maintained by this map
V - the type of mapped values
All Known Subinterfaces:
…, SortedMap<K,V>
All Known Implementing Classes:
… HashMap, Hashtable, …. TreeMap,…

---

public interface **Map<K,V>**

Since:
1.2
See Also:
HashMap, TreeMap, Hashtable, SortedMap, Collection, Set

Nested Class Summary	
static interface	**Map.Entry<K,V>** A map entry (key-value pair).

Method Summary	
void	**clear**() Removes all of the mappings from this map (optional operation).
boolean	**containsKey**(Object key) Returns true if this map contains a mapping for the specified key.
boolean	**containsValue**(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	**entrySet**() Returns a Set view of the mappings contained in this map.
boolean	**equals**(Object o) Compares the specified object with this map for equality.
V	**get**(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
int	**hashCode**() Returns the hash code value for this map.
boolean	**isEmpty**() Returns true if this map contains no key-value mappings.
Set<K>	**keySet**() Returns a Set view of the keys contained in this map.
V	**put**(K key, V value) Associates the specified value with the specified key in this map (optional operation).
void	**putAll**(Map<? extends K,? extends V> m) Copies all of the mappings from the specified map to this map (optional operation).
V	**remove**(Object key) Removes the mapping for a key from this map if it is present (optional operation).
int	**size**() Returns the number of key-value mappings in this map.
Collection<V>	**values**() Returns a Collection view of the values contained in this map.

java.util
Interface Set

All Superinterfaces:
Collection
All Known Subinterfaces:
SortedSet
All Known Implementing Classes:
AbstractSet, HashSet, LinkedHashSet, TreeSet

---

public interface **Set**
extends Collection

A collection that contains no duplicate elements

Since:
1.2
See Also:

Collection, List, SortedSet, HashSet, TreeSet, AbstractSet, Collections.singleton(java.lang.Object), Collections.EMPTY_SET

Method Summary	
boolean	**add**(Object o) Adds the specified element to this set if it is not already present (optional operation).
boolean	**addAll**(Collection c) Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	**clear**() Removes all of the elements from this set (optional operation).
boolean	**contains**(Object o) Returns true if this set contains the specified element.
boolean	**containsAll**(Collection c) Returns true if this set contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this set for equality.
int	**hashCode**() Returns the hash code value for this set.
boolean	**isEmpty**() Returns true if this set contains no elements.
Iterator	**iterator**() Returns an iterator over the elements in this set.
boolean	**remove**(Object o) Removes the specified element from this set if it is present (optional operation).
boolean	**removeAll**(Collection c) Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection c) Retains only the elements in this set that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this set (its cardinality).
Object[]	**toArray**() Returns an array containing all of the elements in this set.
Object[]	**toArray**(Object[] a) Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

java.rmi
Class Naming

java.lang.Object
  └ **java.rmi.Naming**

---

public final class **Naming**
extends Object
Since:
JDK1.1
See Also:
Registry, LocateRegistry, LocateRegistry.createRegistry(int)

Method Summary	
static void	**bind**(String name, Remote obj) Binds the specified name to a remote object.
static String[]	**list**(String name)

	Returns an array of the names bound in the registry.
static Remote	**lookup**(String name) Returns a reference, a stub, for the remote object associated with the specified name.
static void	**rebind**(String name, Remote obj) Rebinds the specified name to a new remote object.
static void	**unbind**(String name) Destroys the binding for the specified name that is associated with a remote object.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

java.rmi.registry.server
class UnicastRemoteObject *partielle*

---

public class **UnicastRemoteObject**
extends RemoteServer

Method Summary	
static Remote	**exportObject**(Remote obj, int port) Exports the remote object to make it available to receive incoming calls, using the particular supplied port.
static boolean	**unexportObject**(Remote obj, boolean force) Removes the remote object, obj, from the RMI runtime.

java.rmi.registry
class LocateRegistry *partielle*

java.lang.Object
  java.rmi.registry.LocateRegistry

---

public final class **LocateRegistry**
extends Object

LocateRegistry is used to obtain a reference to a bootstrap remote object registry on a particular host (including the local host), or to create a remote object registry that accepts calls on a specific port.

Method Summary	
static Registry	**createRegistry**(int port) Creates and exports a Registry instance on the local host that accepts requests on the specified port.
static Registry	**getRegistry**(String host) Returns a reference to the remote object Registry on the specified host on the default registry port of 1099.
static Registry	**getRegistry**(String host, int port) Returns a reference to the remote object Registry on the specified host and port.

java.rmi
interface Remote

---

public interface **Remote**

The Remote interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine. Any object that is a remote object must directly or indirectly implement this interface. Only those methods specified in a "remote interface", an interface that extends java.rmi.Remote are available remotely.

See Also:
UnicastRemoteObject, Activatable

---

java.io
interface Serializable

---

public interface **Serializable**

Serializability of a class is enabled by the class implementing the java.io.Serializable interface. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

During deserialization, the fields of non-serializable classes will be initialized using the public or protected no-arg constructor of the class. A no-arg constructor must be accessible to the subclass that is serializable. The fields of serializable subclasses will be restored from the stream.

---

javax.management
interface NotificationBroadcaster

All Known Subinterfaces:
ModelMBean, ModelMBeanNotificationBroadcaster, NotificationEmitter
All Known Implementing Classes:
MBeanServerDelegate, NotificationBroadcasterSupport, RequiredModelMBean

---

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.

New code should use the NotificationEmitter interface instead.

Method Summary	
void	**addNotificationListener**(NotificationListener listener, NotificationFilter filter, Object handback) Adds a listener to this MBean. If filter is null, no filtering will be performed before handling notifications.
void	**removeNotificationListener**(NotificationListener listener) Removes a listener from this MBean.

javax.management
Interface NotificationListener

All Superinterfaces:
EventListener
All Known Implementing Classes:
RelationService

```
public interface NotificationListener
extends EventListener
```

Should be implemented by an object that wants to receive notifications.

Method Summary	
void	**handleNotification**(Notification notification, Object handback) Invoked when a JMX notification occurs.

javax.management
class NotificationBroadcasterSupport *partielle*

java.lang.Object
  javax.management.NotificationBroadcasterSupport
All Implemented Interfaces:
NotificationBroadcaster, NotificationEmitter

public class **NotificationBroadcasterSupport**
extends Object
implements NotificationEmitter

Provides an implementation of NotificationEmitter interface. This can be used as the super class of an MBean that sends notifications.

Constructor Summary
NotificationBroadcasterSupport()

Method Summary	
void	**sendNotification**(Notification notification) Sends a notification.

javax.management
class Notification ***partielle***

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

public class **Notification**
extends EventObject

The Notification class represents a notification emitted by an MBean. It contains a reference to the source MBean: if the notification has been forwarded through the MBean server, this is the object name of the MBean. If the listener has registered directly with the MBean, this is either the object name or a direct reference to the MBean.

It is strongly recommended that notification senders use the object name rather than a reference to the MBean object as the source.

Constructor Summary
Notification(String type, Object source, long sequenceNumber) Creates a Notification object.

Method Summary	
long	**getSequenceNumber**() Get the notification sequence number.
String	**getType**() Get the notification type.

javax.management.remote
class JMXConnectorFactory ***partielle***

java.lang.Object
  javax.management.remote.JMXConnectorFactory

```
public class JMXConnectorFactory
extends Object
```

Factory to create JMX API connector clients. There are no instances of this class.

Method Summary	
static JMXConnector	**connect**(JMXServiceURL serviceURL) Creates a connection to the connector server at the given address.

javax.management
class MBeanServerInvocationHandler ***partielle***

java.lang.Object
  javax.management.MBeanServerInvocationHandler
All Implemented Interfaces:
InvocationHandler

```
public class MBeanServerInvocationHandler
extends Object
implements InvocationHandler
```

InvocationHandler that forwards methods in an MBean's management interface through the MBean server to the MBean.

Method Summary	
static <T> T	**newProxyInstance**(MBeanServerConnection connection, ObjectName objectName, Class<T> interfaceClass, boolean notificationBroadcaster) Return a proxy that implements the given interface by forwarding its methods through the given MBean server to the named MBean.