Table des matières

* JMX, qu’est ce que c’est?
  + [Généralités](https://blog.jetoile.fr/2010/10/jmx-pour-les-nuls-les-concepts-partie-1.html#generalite)
  + [Architecture JMX](https://blog.jetoile.fr/2010/10/jmx-pour-les-nuls-les-concepts-partie-1.html#architecture)
    - [Niveau instrumentation](https://blog.jetoile.fr/2010/10/jmx-pour-les-nuls-les-concepts-partie-1.html#instrumentation)
    - [Niveau agent](https://blog.jetoile.fr/2010/10/jmx-pour-les-nuls-les-concepts-partie-1.html#agent)
    - [Niveau service distribué](https://blog.jetoile.fr/2010/10/jmx-pour-les-nuls-les-concepts-partie-1.html#distribue)
  + [Composants JMX](https://blog.jetoile.fr/2010/10/jmx-pour-les-nuls-les-concepts-partie-1.html#composant)
    - [MBeans](https://blog.jetoile.fr/2010/10/jmx-pour-les-nuls-les-concepts-partie-1.html#mbean)
    - [Modèle de notifications](https://blog.jetoile.fr/2010/10/jmx-pour-les-nuls-les-concepts-partie-1.html#notification)
    - [Classe de métadonnées de MBeans](https://blog.jetoile.fr/2010/10/jmx-pour-les-nuls-les-concepts-partie-1.html#metadonnee)
    - [Serveur de MBeans](https://blog.jetoile.fr/2010/10/jmx-pour-les-nuls-les-concepts-partie-1.html#serveur)
    - [Service d’agents](https://blog.jetoile.fr/2010/10/jmx-pour-les-nuls-les-concepts-partie-1.html#service)
* Spécifications
  + [JMX Instrumentation](https://blog.jetoile.fr/2010/11/jmx-pour-les-nuls-les-differents-mbeans.html)
    - [MBean](https://blog.jetoile.fr/2010/11/jmx-pour-les-nuls-les-differents-mbeans.html#mbean)
      * [MBean Standard](https://blog.jetoile.fr/2010/11/jmx-pour-les-nuls-les-differents-mbeans.html#mbean_standard)
      * [Dynamic MBean](https://blog.jetoile.fr/2010/11/jmx-pour-les-nuls-les-differents-mbeans.html#mbean_dynamic)
    - [Notification](https://blog.jetoile.fr/2010/11/jmx-pour-les-nuls-les-differents-mbeans.html#notification)
    - [Open MBean](https://blog.jetoile.fr/2010/11/jmx-pour-les-nuls-les-differents-mbeans.html#mbean_open)
    - [Model MBean](https://blog.jetoile.fr/2010/11/jmx-pour-les-nuls-les-differents-mbeans.html#mbean_model)
  + [Agent JMX](https://blog.jetoile.fr/2010/11/jmx-pour-les-nuls-les-agents-jmx-partie.html#agent)
  + [Concepts](https://blog.jetoile.fr/2010/11/jmx-pour-les-nuls-les-classes-de-base.html)
    - [ObjectName](https://blog.jetoile.fr/2010/11/jmx-pour-les-nuls-les-classes-de-base.html#objectName)
    - [ObjectInstance](https://blog.jetoile.fr/2010/11/jmx-pour-les-nuls-les-classes-de-base.html#objectInstance)
    - [Attribute et AttributeList](https://blog.jetoile.fr/2010/11/jmx-pour-les-nuls-les-classes-de-base.html#attribute)
    - [Les Exceptions](https://blog.jetoile.fr/2010/11/jmx-pour-les-nuls-les-classes-de-base.html#exception)
  + [MBean Server](https://blog.jetoile.fr/2010/11/jmx-pour-les-nuls-le-mbean-server.html#mbean_server)
  + [Chargement dynamique des MBeans](https://blog.jetoile.fr/2010/12/jmx-pour-les-nuls-chargement-dynamique.html#mbean_dynamic)
  + [Les services JMX](https://blog.jetoile.fr/2010/12/jmx-pour-les-nuls-les-services-jmx.html)
    - [Service Monitoring](https://blog.jetoile.fr/2010/12/jmx-pour-les-nuls-les-services-jmx.html#monitoring)
    - [Service Timer](https://blog.jetoile.fr/2010/12/jmx-pour-les-nuls-les-services-jmx.html#timer)
    - [Service Relation](https://blog.jetoile.fr/2010/12/jmx-pour-les-nuls-les-services-jmx.html#relation)
    - [Service Sécurité](https://blog.jetoile.fr/2010/12/jmx-pour-les-nuls-les-services-jmx.html#securite)
  + [Les Connecteurs](https://blog.jetoile.fr/2010/12/jmx-pour-les-nuls-les-connecteurs.html#connector)

JMX, qu’est ce que c’est?

Généralités

*Java Management eXtensions* (JMX) définit une architecture, un design pattern, une API et les services permettant de superviser et d’administrer une application au travers du langage Java.

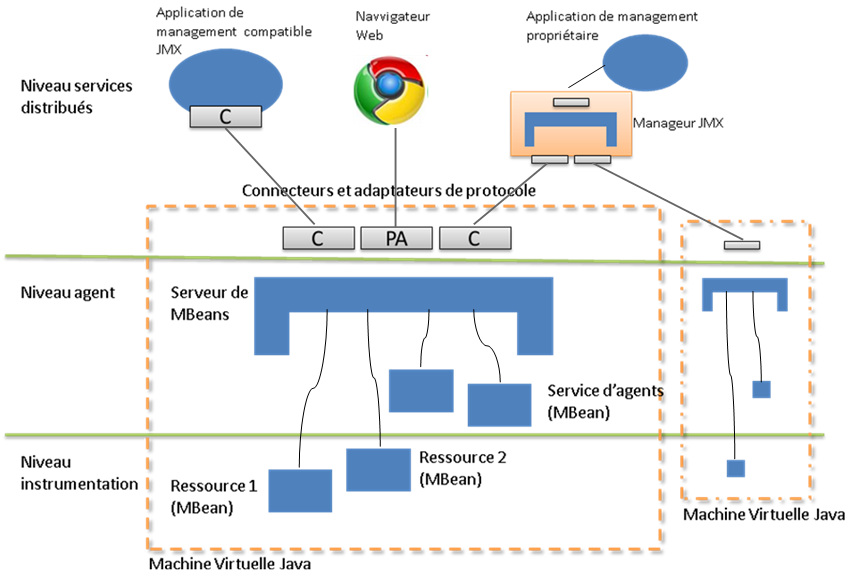
JMX permet, en effet :

* d’administrer et de superviser une application Java sans avoir à investir dans de couteux serveurs : l’architecture JMX repose sur un serveur d’objets de base qui peut être vu comme un agent de supervision et d’administration et qui peut fonctionner sur presque toutes les JVM.
* d’offrir une architecture de supervision et d’administration scalable : chaque agent JMX est un module indépendant pouvant se brancher à l’agent de supervision et d’administration de l’application à surveiller.
* de s’interfacer avec une autre solution de supervision et d’administration : les agents JMX peuvent être supervisés et administrés aux travers de différents protocoles (HTTP, SNMP, …) et l’API JMX permet de définir le sien.
* de s’intégrer avec les autres technologies Java présentes dans le système telles que JNDI, JDBC, JINI, SLP, …

Architecture JMX

L’architecture JMX peut se décomposer en trois niveaux :

* le niveau instrumentation (**instrumentation level**),
* le niveau agent (**agent level**),
* et le niveau service distribué (**distributed services level**).



Niveau instrumentation

Le niveau instrumentation spécifie comment implémenter des ressources administrable et supervisable par JMX ; une ressource pouvant être une application, une implémentation d’un service, un périphérique, ou autre tant qu’elle est développée en Java (ou qu’un wrapper existe) et qu’elle a été instrumentée pour être compatible JMX.

L’instrumentation d’une ressource donnée est fournit par la présence d’un ou plusieurs Beans supervisables et administrables (ou MBeans pour *Managed Beans*) qui peuvent être de plusieurs sortes :

* **MBean standard**,
* **MBean dynamique**.

Ainsi, l’instrumentation d’une ressource permet de la rendre administrable et supervisable au travers du niveau agent. Cela permet aux MBean de s’abstraire du niveau agent et ainsi que de les rendre plus simple et évolutifs.

Niveau agent

Le niveau agent spécifie comment implémenter les agents qui permettent le contrôle direct des ressources et qui les rend administrable et supervisable d’une application de supervision et d’administration distante. Les agents se trouvent être généralement sur la même machine que les ressources qu’elles contrôlent mais cela n’est pas un pré-requis.

Ce niveau s’appuie et utilise le niveau d’instrumentation pour définir des agents standards utilisés pour se superviser et s’administrer elle-même.

L’**agent JMX** est constitué d’un serveur de MBeans (**MBeanServer**) et d’un ensemble de services permettant de manipuler les MBeans. En outre, le MBeanServer nécessite au moins un connecteur ou adaptateur. L’agent JMX peut être embarqué dans la machine virtuelle Java qui accueille les ressources JMX à administrer et superviser, ou peut être instancié dans un élément de médiation quand la ressource n’est pas une ressource Java.

Enfin, l’agent JMX n’a pas besoin de connaitre les ressources qui doivent être administrées et supervisées. Le manageur accède donc au MBeans de l’agent et utilise les services fournis par ce dernier au travers un adaptateur de protocoles ou un connecteur. Celui permet à l’agent JMX d’être indépendant et faiblement couplé à l’application de management.

Niveau service distribué

Le niveau service distribué spécifie comment implémenter l’interface d’administration et de supervision JMX. Ce niveau permet de définir les interfaces d’administration et de supervision et les composants qui exploitent les agents ou la hiérarchie d’agents JMX. Ces composants peuvent :

* fournir une interface d’administration et de supervision des applications via leurs agents JMX au travers d’un connecteur,
* exposer une vue d’administration et de supervision d’un agent JMX et de ses MBeans en associant leur signification sémantique avec un protocole haut niveau (comme HTML ou SNMP), diffuser les informations de supervision d’une plate-forme de supervision et d’administration à différents agents JMX,
* agréger et consolider les informations de supervision et d’administration en provenance de différents agents JMX dans une vue logique à destination de l’utilisateur final,
* gérer et sécuriser l’administration et la supervision du système.

En fait, les composants d’administration et de supervision peuvent interagir les uns les autres au sein du réseau afin de fournir des fonctions d’administration et de supervision scalables et distribuées.

Composants JMX

JMX s’appuie sur les composants et notions suivants :

* **MBeans** (standard, dynamic, open et model)
* **Modèle de notifications** (Notification Model)
* **Classe de métadonnées de MBeans** (MBean metadata classes)
* **Serveur de MBeans** (MBean Server)
* **Services d’agents** (Agent Services)

MBeans

Un MBean est un objet Java qui implémente une interface spécifique et qui respecte un design pattern particulier. Cela permet de normaliser la représentation de l’interface des ressources à gérer dans le MBean. Cette dernière est composée de l’ensemble des informations et des méthodes nécessaires à l’administration et à la supervision de l’application.

L’interface de gestion d’un MBean peut être composée :

* d’attributs dont les valeurs peuvent être accédées,
* d’opérations qui peuvent être invoquées,
* de notifications qui peuvent être émises,
* de constructeurs (au sens Java du terme)

En fait, un MBean encapsule les attributs et les opérations au travers de leurs méthodes publiques : par exemple, un attribut en lecture-seule dans un MBean standard n’aura qu’un *getter* alors qu’un attribut accessible en lecture-écriture disposera d’un *getter* et d’un *setter*.

Ainsi, n’importe quel objet qui implémentera un MBean et qui sera déclaré auprès de l’agent JMX pourra être géré.

Les autres types de composants JMX comme les services d’agents sont des MBeans complètement instrumentés leur permettant de bénéficier de l’infrastructure JMX.

La spécification JMX définit quatre types de MBean :

* **Standard MBean** : c’est l’implémentation de MBean la plus simple. Elle fournit une représentation statique des ressources à gérer et à superviser. Son interface de gestion est décrite par le nom de ses méthodes.
* **Dynamic MBean** : ce MBean doit implémenter une interface spécifique mais il expose ses interfaces de gestion de manière dynamique à l’exécution.
* **Open MBean** : ce MBean est une extension du Dynamic MBean mais repose sur une représentation normée de ses types de données afin de permettre plus d’interopérabilité. Pour ce faire, il s’auto-décrit.
* **Model MBean** : ce MBean est une extension du Dynamic MBean mais est complètement configurable et s’auto-décrit à l’exécution. Il est à noter qu’il fournit une classe générique de MBean qui définit son comportement par défaut qu’il est possible de redéfinir.

Modèle de notifications

JMX définit un modèle de notifications génériques qui se base sur le modèle d’événements Java. Les notifications peuvent être émises par une instance de MBean mais aussi par le serveur de MBean lui-même.

Classe de métadonnées de MBeans

JMX définit comment les classes doivent décrire les interfaces de gestion des MBean. Ces dernières servent à construire la structure des informations à publier qui est utilisée par le serveur de MBeans. Ainsi, ces classes de métadonnées contiennent la structure qui décrit tout ce dont il a besoin : ses attributs, ses opérations, ses notifications et ses constructeurs mais aussi les caractéristiques de chacun de ces éléments comme la signature des méthodes ou le niveau d’accessibilité des attributs.

Serveur de MBeans

Le serveur de MBean peut être vu comme un registre des objets qui sont gérables au travers de l’agent. Ainsi, chaque objet enregistré dans le serveur de MBean est visible de l’application de gestion.

Cependant, il n’expose jamais les objets directement mais seulement leurs interfaces même s’il fournit une interface standardisée d’accès aux MBeans présent dans la même JVM.

Un MBean doit donc être enregistré de manière unique au sein du serveur de MBean et peut l’être par :

* un autre MBean,
* l’agent lui-même,
* une application de gestion distante au travers d’un service distribué.

Les opérations alors disponibles sur un MBean sont :

* la découverte de l’interface de gestion du MBean,
* la lecture et l’écriture de la valeur des attributs,
* l’invocation des opérations offertes par l’interface de gestion du MBean,
* la récupération des notifications émises par le MBean.

Pour permettre à un agent d’être accessible depuis une application extérieure, le MBean Server s’appuie sur les adaptateurs de protocoles et les connecteurs.

Service d’agents

Les services d’agents sont les objets qui gèrent les différentes opérations de gestion des MBean enregistrés dans le serveur de MBeans.

Ces services d’agents sont généralement eux-mêmes des MBeans et permettent ainsi au serveur de MBeans de les gérer.

Par défaut, doivent être présents les services suivants :

* **Management Applet** (m-let) qui permet le chargement dynamique de classes
* **Service Timers** qui permet de déclencher des actions sur les MBeans de façon régulière
* **Service Relation** qui permet d’associer des MBeans entre eux
* **Service Monitor** qui permet de scruter certaines valeurs d’attribut afin de notifier d’autres objets lors d’un changement

Le mot de la fin de cette partie

Dans cet article, nous avons vu (ou revu) les concepts de bases de JMX. Dans les articles suivants, nous entrerons plus dans les entrailles des spécifications en nous calquant sur les spécifications JMX pour tenter d’y voir un peu plus clair…

MBean

Un MBean est une classe java concrète qui :

* doit implémenter sa propre interface MBean ou implémenter l’interface DynamicMBean,
* peut implémenter l’interface NotificationBroacaster

Une classe qui implémente sa propre interface de MBean est un MBean Standard. Cette manière de procéder est la plus simple. Dans ce cas, les opérations et attributs accessibles sont déterminés statiquement grâce à l’introspection de l’interface du MBean.

Pour un Dynamic MBean (et donc une classe implémentant l’interface DynamicMBean), les opérations et les attributs sont exposés de manière indirecte à l’agent JMX qui invoque alors des méthodes pour déterminer le nom et la nature des opérations et attributs.

Un MBean n’est pas obligatoirement une classe publique Java. Cependant, elle doit implémenter une interface publique (que ce soit ses propres opérations pour un MBean standard ou l’interface DynamicMBean pour un MBean dynamique).

En outre, le MBean peut avoir plusieurs constructeurs qui doivent être publiques s’il doit être instancié par un agent.

MBean Standard

Comme il a été vu précédemment que, pour être gérable via un agent JMX, un MBean Standard doit explicitement définir son interface de gestion et doit être conforme à certaines règles de nommage qui sont inspirées du modèle des JavaBean.

Ainsi, l’interface de gestion d’un MBean Standard peut être composée de :

* ses constructeurs : seuls les constructeurs publics sont exposés,
* ses attributs : les propriétés qui sont exposées le sont au travers des getter et setter,
* ses opérations : les méthodes qui sont exposées sont toutes les méthodes qui ne sont pas des getter ou des setter,
* ses notifications.

En outre, l’interface qui est implémenté par le MBean doit porter le même nom que ce dernier mais avec le suffixe MBean(par exemple, si la classe à gérer se nomme MyClass, alors l’interface devra se nommer MyClassMBean).

Une classe peut aussi être administrable par son héritage si elle respecte un des modèles suivants :

| **Modèles** | **Commentaires** |
| --- | --- |
| small | Les opérations exposées sont a1 et a2. |
| small | Si B étend A sans implémenter sa propre interface, alors B hérite de l’interface de management AMBean et les opérations exposées sont a1 et a2. |
| small | B hérite de A et implémente sa propre interface de management BMBean. L’opération exposée est b2. |
| small | B hérite de A et implémente son interface de management qui hérite de celle de A. Les opérations exposées sont a1, a2 et b2. A noter que si B n’étendait pas A, les opérations exposées seraient similaires. |
| small | B implémente son interface de management qui hérite de l’interface AMBean. Les opérations exposées sont a1, a2 et b2. |

Les attributs sont les champs ou les propriétés des MBean implémentés (ou étendus). Ils sont déterminés par leurs noms (la méthode Character.isJavaIdentifierStart (resp. Character.isJavaIdentifierPart) doit renvoyer vrai pour sa première lettre (resp. ses autres lettres)) et par leurs types et sont accédés au travers de leurs accesseurs. Aussi, si un attribut n’a pas d’accesseurs, il ne sera pas visible puisque, pour rappel, ses accesseurs permettent également de déterminer son niveau d’accessibilité (lecture seule, écriture seule ou lecture-écriture). Cependant, il est à noter qu’un attribut ne doit avoir qu’un seul *getter* et/ou *setter* et peut être de n’importe quel type Java (cela inclus également les tableaux).

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Un attribut doit suivre la règle de nommage suivante :   * Sa première lettre doit renvoyée vrai à la méthode *Character.isJavaIdentifierStart* * Ses autres lettres doivent renvoyées vrai à la méthode *Character.isJavaIdentifierPart* |

|  |  |  |  |
| --- | --- | --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Les attributs et opérations sont case sensitives. En outre, un attribut ne peut avoir qu’un seul getter et/ou setter. Par exemple, les attributs suivants ne sont pas valides :   |  |  | | --- | --- | | 1  2  3  4  5  6 | boolean valid ;  int number ;  boolean isValid() {return this.valid ;}  boolean getValid() {return this.valid ;}  void setNumber(int number) {this.number = number ;}  void setNumber(String number) {this.number = Integer.valueOf(number) ;} | |

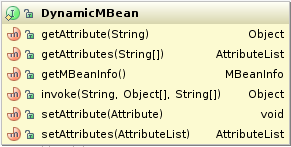
Les opérations JMX peuvent avoir différentes fonctions (comme effectuer une opération interne sur la ressource managée ou renvoyer une valeur) et sont des méthodes Java qui doivent être spécifiées dans l’interface de management (MBean). Ainsi, toute opération qui ne répond pas au design pattern défini par la définition d’attribut JMX est vue comme une opération.

Dynamic MBean

Alors que les MBean Standard permettent de gérer des ressources bien définies, les Dynamics MBean offrent une solution plus flexible. En effet, les Dynamics MBean sont des ressources qui sont instrumentés au travers d’une interface prédéfinie qui n’expose les attributs et les opérations qu’à l’exécution. En effet, au lieu d’exposer ses opérations et attributs au travers de méthodes déterminées à l’avance, les Dynamics MBean implémentent une méthode qui retourne tous ses attributs ainsi que la signature de toutes ses méthodes, permettant ainsi de rendre l’exposition dynamique.

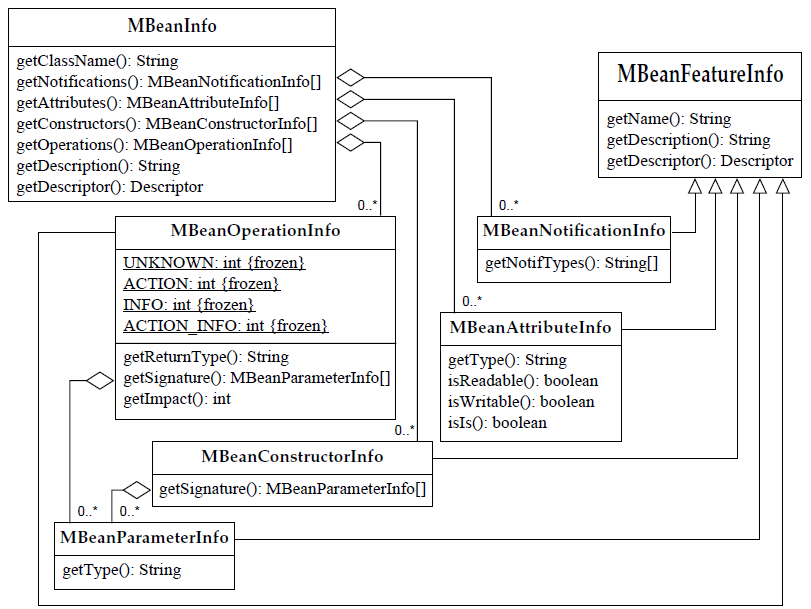
En fait, un MBean qui implémente l’interface DynamicMBean ne fonctionne pas par introspection comme les MBean Standard mais appelle des méthodes qui lui renvoient les attributs et opérations gérables.

Ainsi, un MBean Dynamic est une classe Java qui implémente directement ou indirectement (ie. par héritage) l’interface publique DynamicMBean.



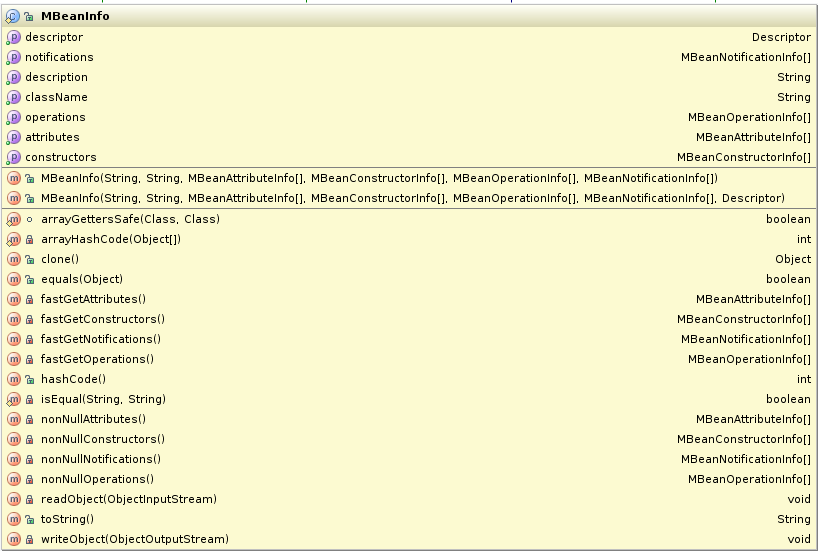
En fait, cette interface définie les méthodes suivantes :

* getMBeanInfo qui retourne une instance d’un MBeanInfo et qui contient la définition de l’interface de management du MBean, c’est-à-dire la liste des ses attributs associés à leurs types et leurs propriétés (lecture-seule, écriture, lecture-écriture), de ses opérations associées à leurs signatures, la liste des notifications ainsi que la description des constructeurs offerts par le MBean.

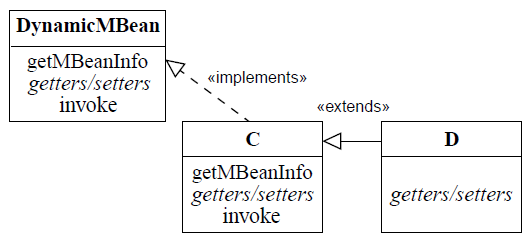


* getAttribute et getAttributes qui retournent le ou les attributs demandés.
* setAttribute et setAttributes qui permettent de renseigner les attributs.
* invoke qui permet d’invoquer toutes les opérations offertes par le MBean.

En outre, tout comme pour les MBean Standard, les Dynamic MBean peuvent hériter leur instrumentation d’une superclasse, mais il est à noter que l’interface de management ne peut être composée d’un arbre d’héritage puisque l’interface de management est fournie par le MBeanInfo.



Ainsi, dans l’exemple suivant, si la classe D étend la classe C sans redéfinir la méthode getMBeanInfo, alors D aura la même interface de management que C. Cependant, D pourra, en redéfinissant les accesseurs redéfinir l’implémentation de ses attributs.



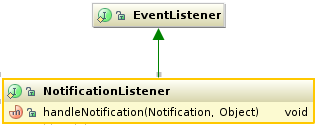
|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Le serveur MBean ne teste ni ne valide la description du Dynamic MBean : c’est au développeur de vérifier si l’interface de management est conforme à l’implémentation interne. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | De par sa nature, un Dynamic MBean est susceptible d’évoluer à l’exécution de l’application. Cependant, seules des applications de management propriétaires peuvent prendre en compte ces changements puisqu’il est nécessaire de les notifier en cas de changements. |

Notification

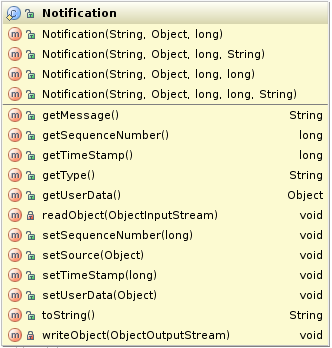
Les interfaces de gestion des MBean permettent à un agent de récupérer ou de modifier la valeur des attributs des ressources ainsi que d’invoquer des opérations sur ces dernières. Cependant, JMX permet également de notifier les applications de gestion en cas de changement d’état du système (et plus particulièrement des ressources).

Ainsi, JMX propose un modèle pour émettre des événements (appelés **Notification**) fonctionnant sur le principe d’écouteurs : les applications de managements et les autres objets enregistrés peuvent s’enregistrer comme des écouteurs (NotificationListener) du MBean émetteur de notifications (Notification). Il est cependant à noter qu’il ne leur est possible de ne s’enregistrer qu’une seule fois auprès de l’émetteur.

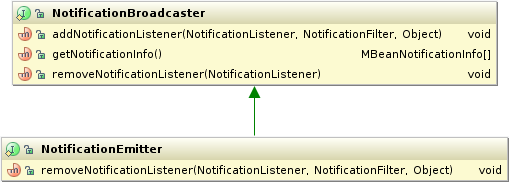


Les notifications sont encapsulées dans une implémentation de l’interface Notification qui est composée :

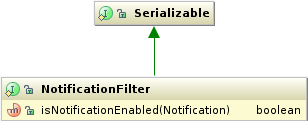
* du type de notification (de type String) qui est formatée à la façon des propriétés java (ie. une chaine de caractère utilisant des points. Par exemple : vendorName.resourceA.eventA1). Il est à noter que les types de notification préfixés par JMX. sont réservées aux notifications émises par l’infrastructure JMX (par exemple, JMX.mbean.registered).
* d’un numéro de séquence qui permet d’identifier une instance particulière de notification dans un contexte de broadcast.
* du timestamp qui permet de savoir quand la notification a été générée.
* du message émis (de type String) qui peut permettre d’expliciter la notification à l’utilisateur.
* d’information utilisateur (de type Object) qui peut contenir d’autres types d’information.



Du coté de l’émetteur, il doit implémenter soit l’interface NotificationBroadcaster, soit l’interface NotificationEmitter.



En outre, il est également possible de filtrer les notifications reçues en implémentant l’interface NotificationFilter.

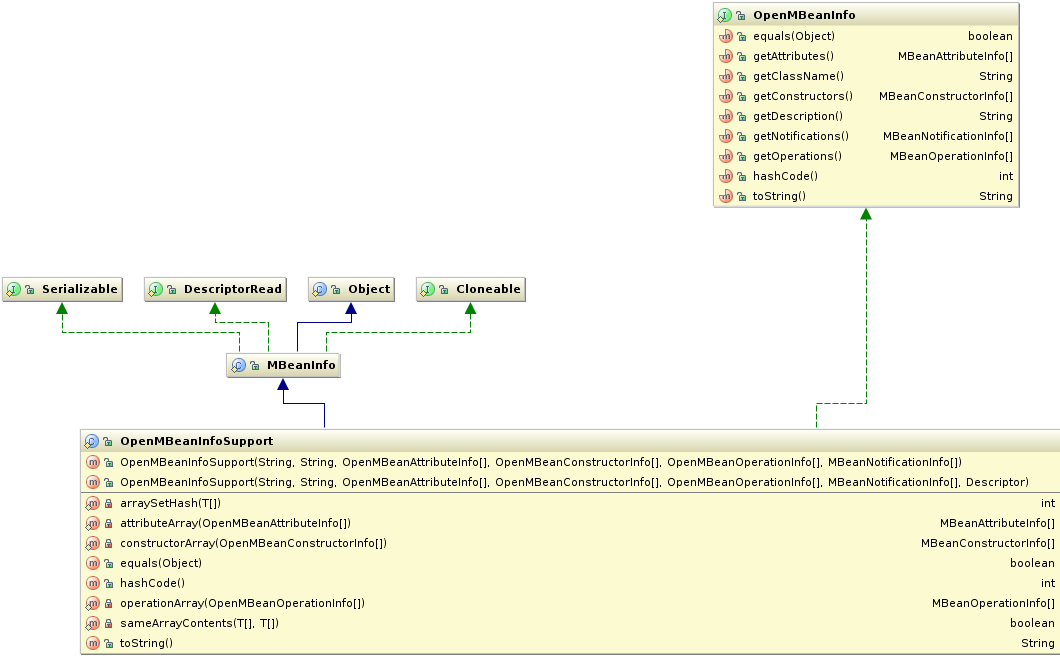


Open MBean

Les Open MBean sont un-sous type des Dynamic MBean. En fait, il s’agit de Dynamic MBean mais dont le type des attributs et des signatures des opérations a été normalisé afin de permettre une meilleure interopérabilité entre le MBean et l’interface de management. Ainsi, l’application de management et les Open MBean peuvent partager et utiliser les données de management (attribut) et invoquer les opérations sans nécessiter une recompilation, un ré-assemblement ou un linkage dynamique, opérations qui nécessiteraient que l’application de management ait accès aux classes Java de l’agent si les types n’étaient pas normalisés.

Cependant, les Open MBean ne sont que des Dynamic MBean, c’est-à-dire que c’est à la charge du développeur de vérifier qu’ils n’utilisent que les types définis supportés par les Open MBean.

Aussi, la méthode getMBeanInfo() doit retourner une instance d’un OpenMBeanInfoSupport plutôt qu’une instance d’un MBeanInfo (pour rappel, un Dynamic MBean doit implémenter l’interface DynamicMBean qui possède la méthode getMBeanInfo() et qui renvoie un objet de type MBeanInfo).



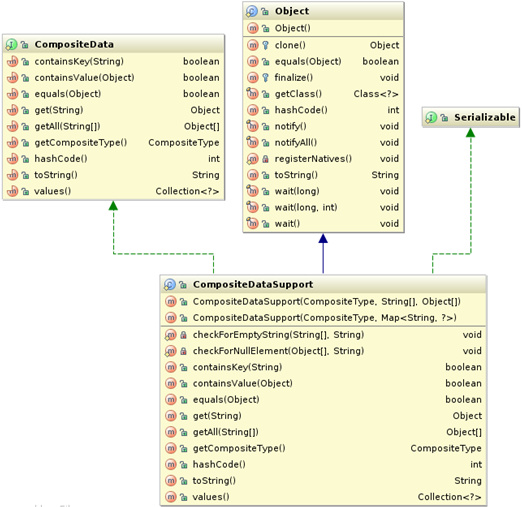
Les types supportés par les Open MBean sont :

|  | **Type de données sypportées par les Open MBean** |  |
| --- | --- | --- |
|  | java.lang.Boolean | java.lang.Float | |
|  | java.lang.Byte | java.lang.Integer | |
|  | java.lang.Character | java.lang.Long | |
|  | java.lang.Double | java.lang.Short | |
|  | boolean[] | float[] | |
|  | byte [] | int[] | |
|  | char[] | long[] | |
|  | double[] | short[] | |
|  | java.lang.String | java.lang.Void (retour d’opérations seulement) | |
|  | java.math.BigDecimal | java.math.BigInteger | |
|  | java.util.Date | javax.management.ObjectName | |
|  | javax.management.openmbean.CompositeData (interface) | javax.management.openmbean.TabularData (interface) | |

Les types CompositeData et TabularData sont utilisés pour agréger les données basiques (tels que les String ou les tableaux de int) et fournissent un mécanisme pour fournir des structures de données complexes. Bien que ces types soient des types simples (au sens JMX du terme), elles peuvent contenir d’autres CompositeData ou TabularData.

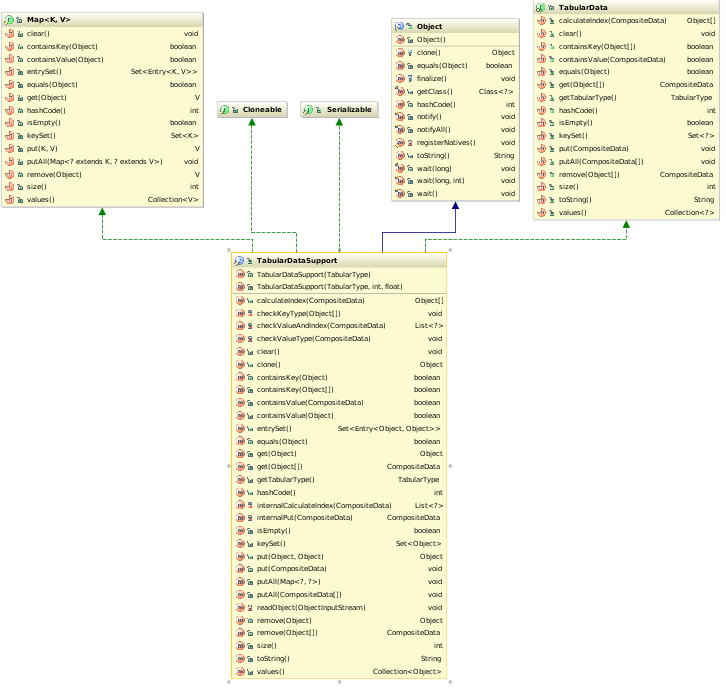
En fait, une implémentation d’un CompositeData est équivalente à une Map, alors qu’une instance d’un TabularData est équivalente à un tableau de CompositeData.

Il est à noter qu’un CompositeData est une structure immuable. Ainsi dès qu’il est instancié, il n’est pas possible de le modifier. Le TabularData peut, quant à lui, être modifié.

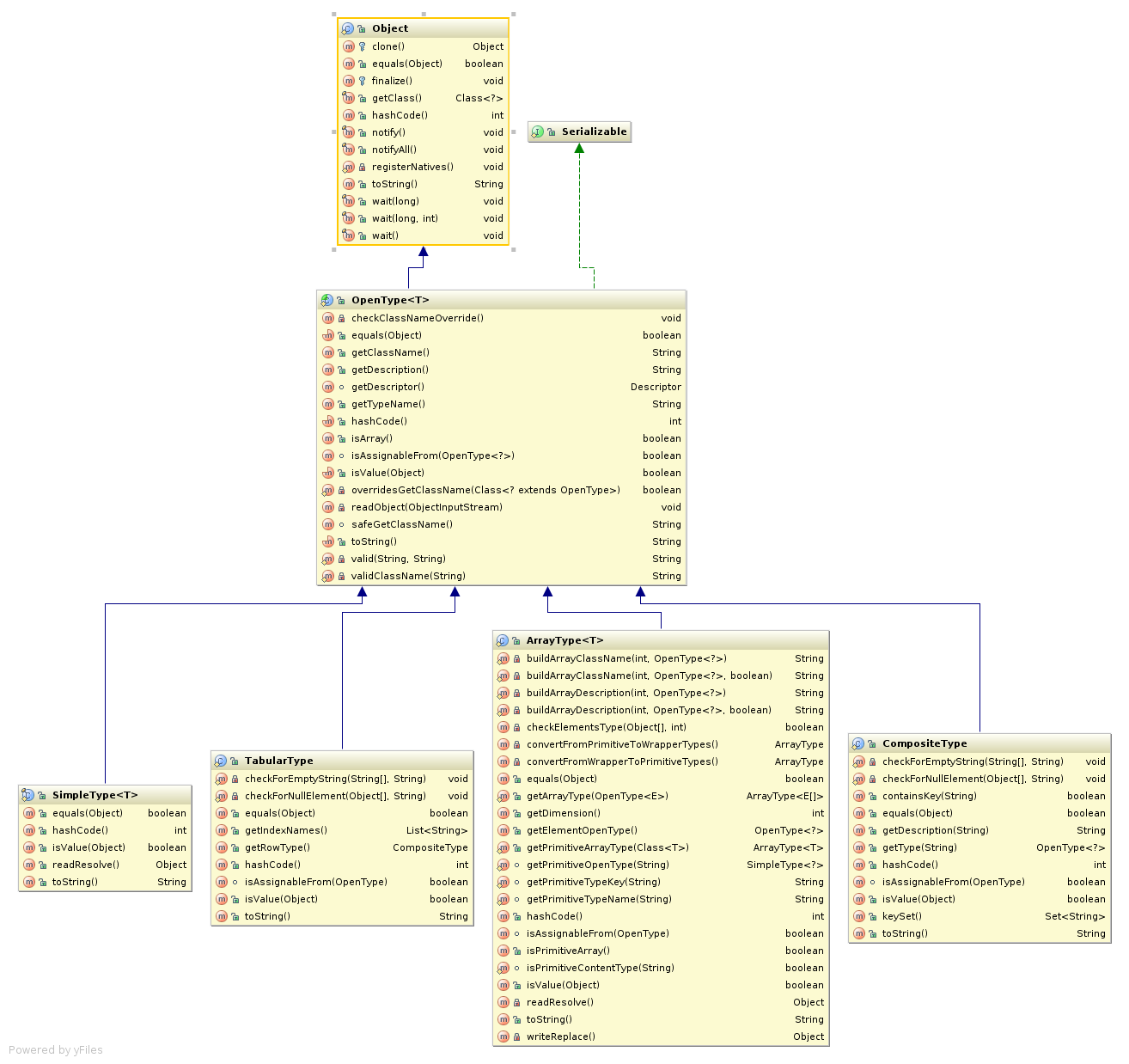


JMX fournit une implémentation de ces interfaces qu’il est possible d’utiliser. Pour le CompositeData, il s’agit de CompositeDataSupport et, pour le TabularData, TabularDataSupport (classe qui définie une structure de tableau contenant des CompositeData qui doivent avoir le même CompositeType).

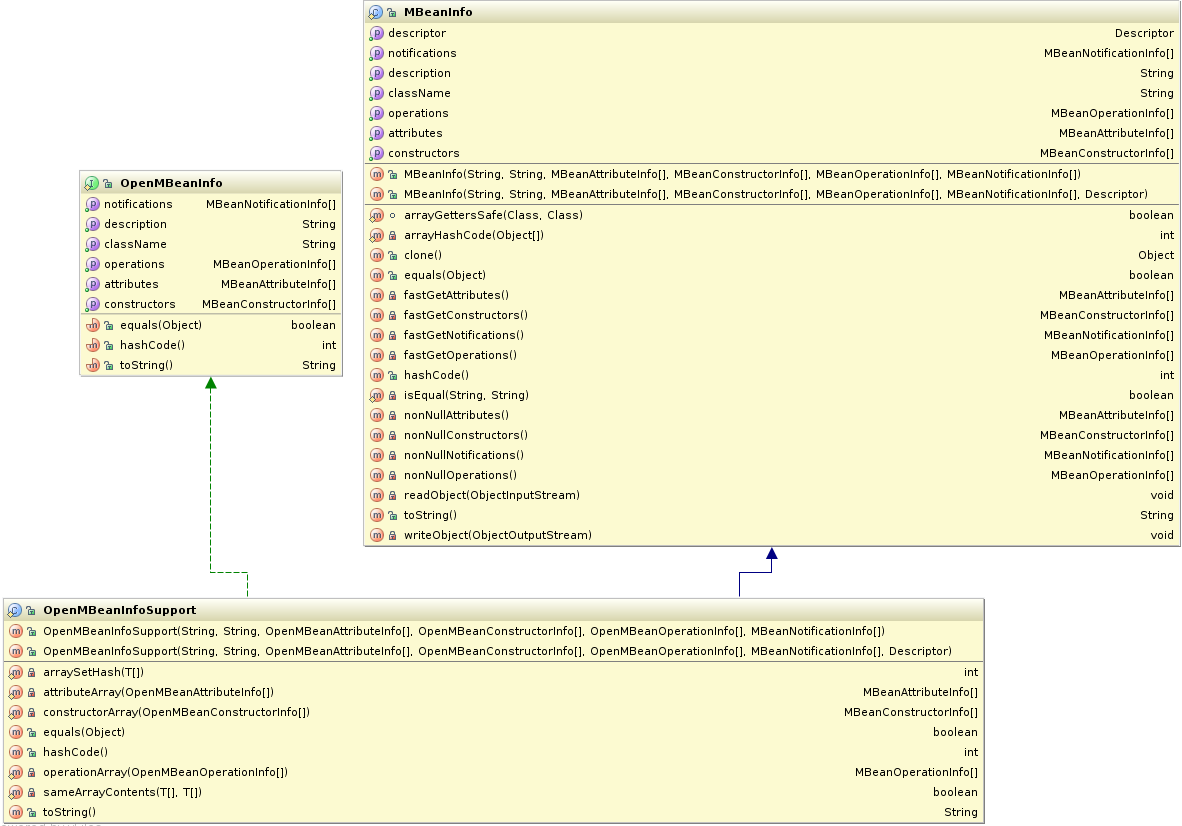
|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Les classes GCInfo, MemoryNotifInfoCompositeData, MemoryUsageCompositeData, MonitorInfoCompositeData, StackTraceElementCompositeData, ThreadInfoCompositeData et VMOptionCompositeData implémentent également l’interface CompositeDataSupport. |

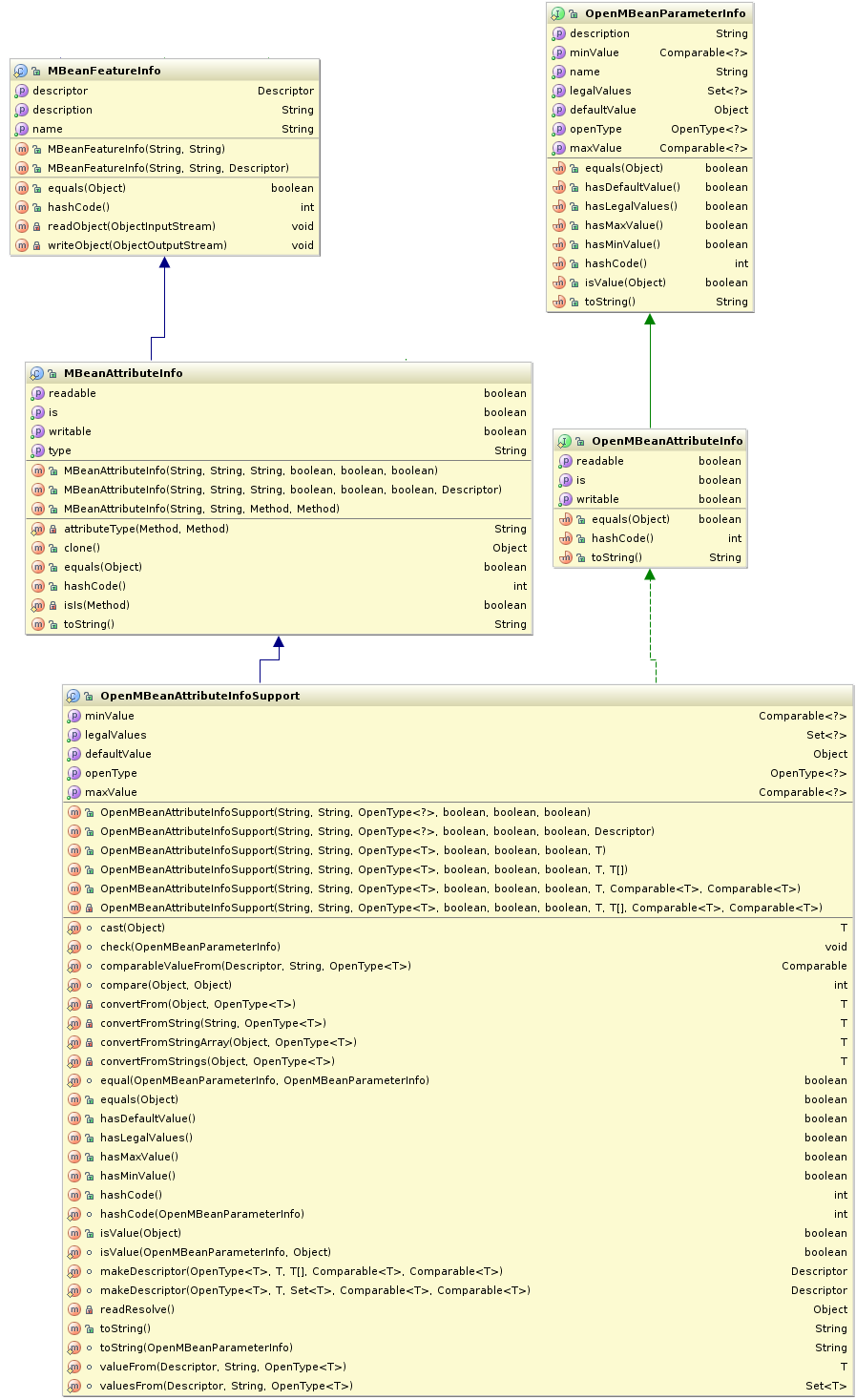


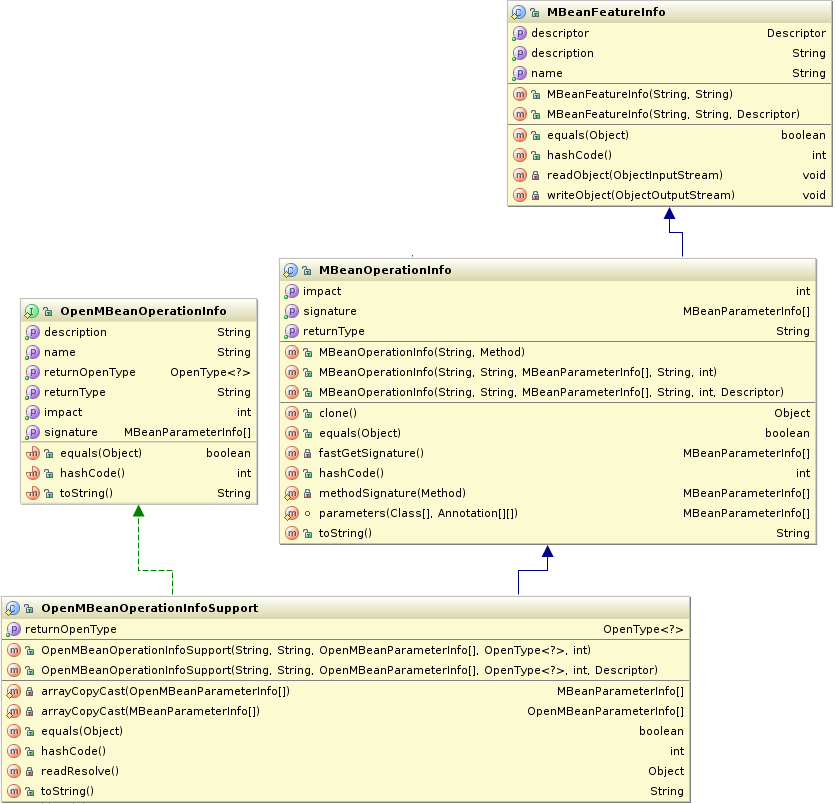
Cependant, le fait d’utiliser des types de données complexes nécessite que l’Open MBean doit décrire la structure des données. Pour cela, les Open MBean proposent une classe abstraite OpenType qui décrit aussi bien ses types basiques que ses types complexes. Cette classe abstraite est donc étendue par des classes qui décrivent chacun de ses types cités ci-dessus en spécifiant le nom du type, sa description ainsi que la classe qui la spécifie. Cela permet, pour les types complexes, de fournir une description et un nom de chaque élément qui le compose.

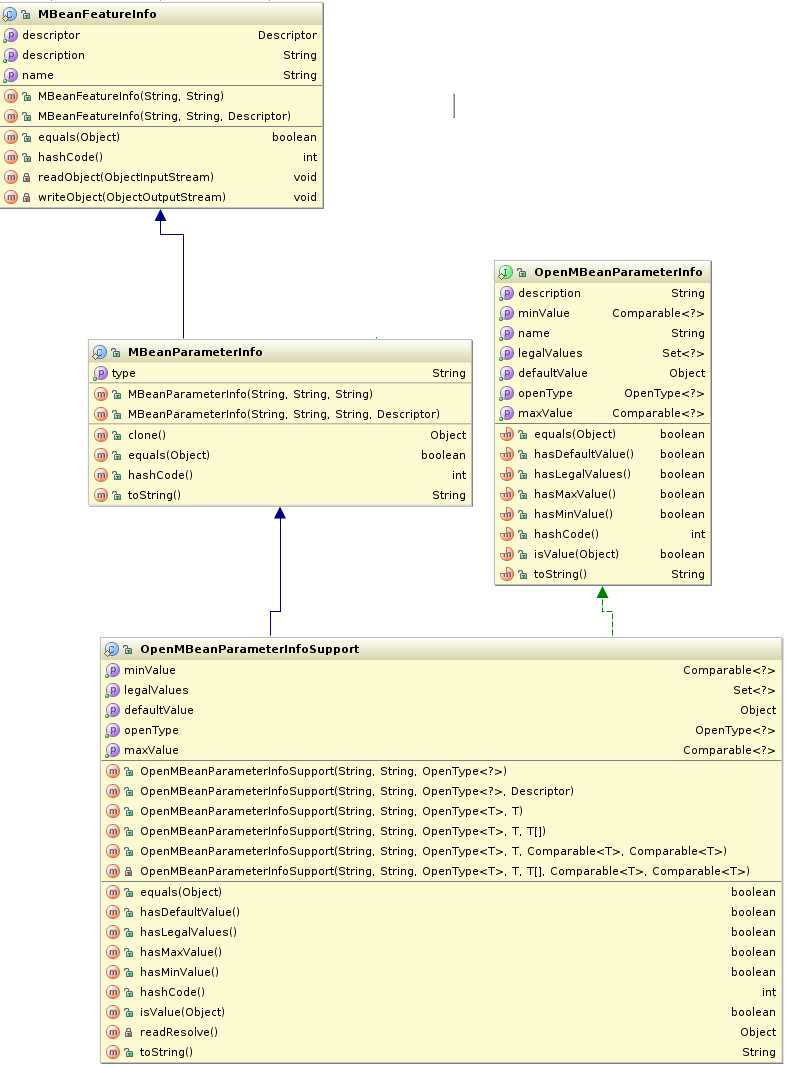


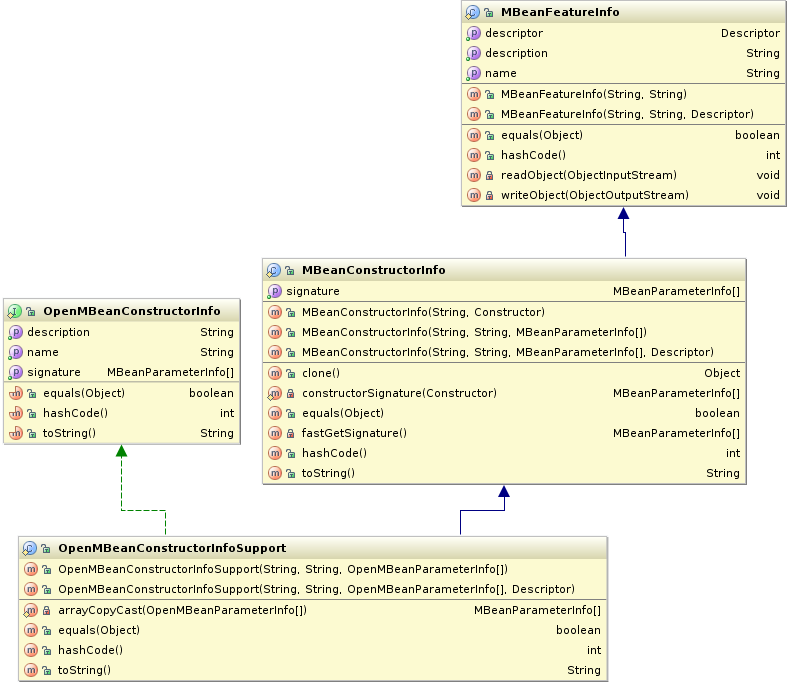
De plus, comme indiqué précédemment, un Open MBean étant un Dynamic MBean, il doit s’auto décrire via l’interface MBeanInfo. Pour cela, JMX propose, par défaut, des sous-classes des classes permettant de décrire un Dynamic MBean (MBeanInfo, MBeanAttributeInfo, MBeanOperationInfo, MBeanParameterInfo, MBeanConstructorInfo) et dont le nom est préfixé par Open et postfixé par Support (OpenMBeanInfoSupport, OpenMBeanAttributeInfoSupport, OpenMBeanOperationInfoSupport, OpenMBeanParameterInfoSupport, OpenMBeanConstructorInfoSupport).

Les notifications sont, quant à elle, standard.







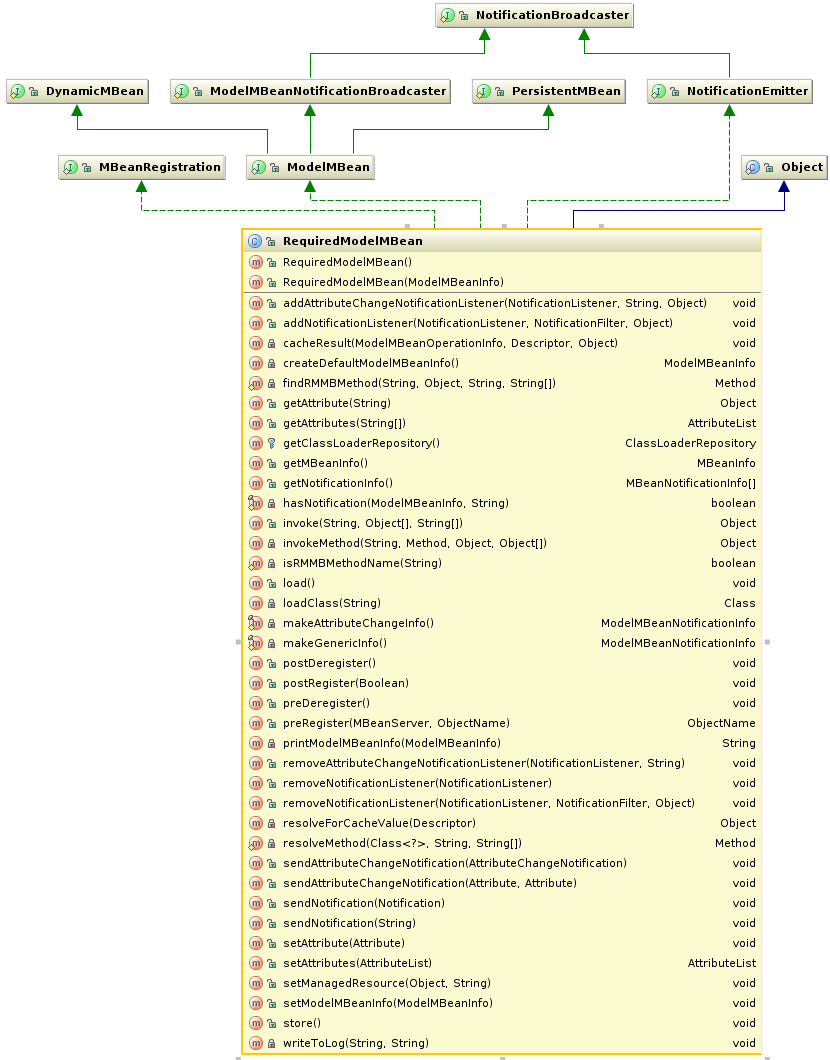


|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Pour développer un Open MBean, il faut :   * Que la ressource à manager implémente l’interface *DynamicMBean* * Que tous les attributs et les signatures des méthodes soient des types supportés par les OpenMBean * Que l’implémentation de la méthode *getMBeanInfo* retourne une instance de l’interface *OpenMBeanInfo* (par exemple, une instance d’un *OpenMBeanInfoSupport*) qui doit retourner des objets de type *OpenMBeanXXXInfo* * Que les méthodes suivantes retournent des objets valides et non null :   + *OpenMBeanInfo.getDescription*   + *OpenMBeanOperationInfo.getDescription*   + *OpenMBeanConstructorInfo.getDescription*   + *OpenMBeanParameterInfo.getDescription*   + *OpenMBeanAttributeInfo.getDescription*   + *MBeanNotificationInfo.getDescription* * Que la méthode *OpenMBeanOperationInfo.getImpact* retourne une des constantes suivantes :   + *ACTION*,   + *INFO*,   + *ACTION\_INFO*. |

Model MBean

Un Model MBean est un MBean générique configurable ayant pour objectif de fournir un patron de MBean pouvant être utilisé simplement par n’importe quelles ressources. Il s’agit, en fait, d’un MBean Dynamic particulier dont l’interface définie une structure qui, lorsqu’elle est implémentée, fournie un MBean avec un comportement par défaut. Ces Model MBean devant être supportés par les agents JMX, ils peuvent être utilisés par n’importe quels applications, ressources et services pour créer un objet gérable à l’exécution : les utilisateurs n’ont qu’a instancier un Model MBean, configurer son comportement par défaut et l’enregistrer au sein de l’agent JMX.

En fait un Model MBean est constitué d’un ensemble d’interfaces et de classes concrètes fournit par l’agent JMX (qui doit fournir une implémentation de la classe RequiredModelMBean et qui a pour objectif de fournir un comportement par défaut).



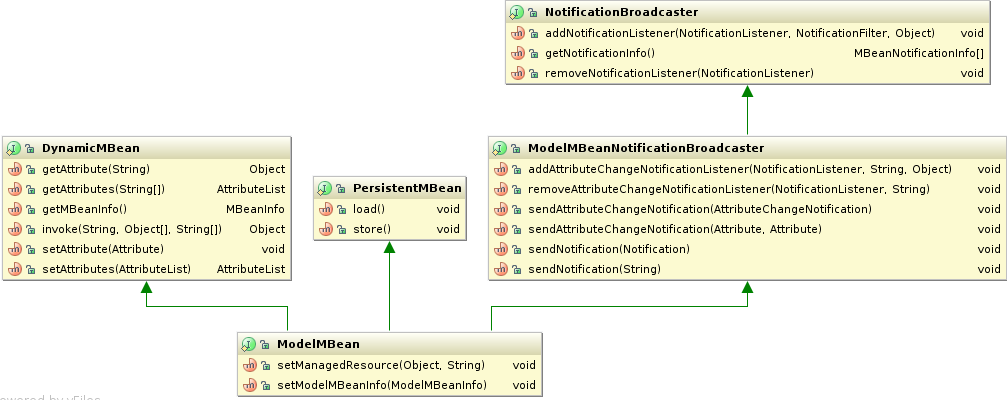
Un MBean Server est donc un repository et factory de Model MBean qui peuvent être obtenus au travers de l’agent JMX afin de rendre une ressource gérable : le développeur n’a pas besoin de fournir une implémentation de cette classe mais juste la configurer à l’exécution afin d’exposer l’interface d’administration et de supervision nécessaire à la ressource.

La ressource à administrer et à superviser ajoute ses attributs, ses opérations et ses notifications à l’objet basique Model MBean en s’interfaçant avec l’agent JMX et à son ou ses Model MBean.

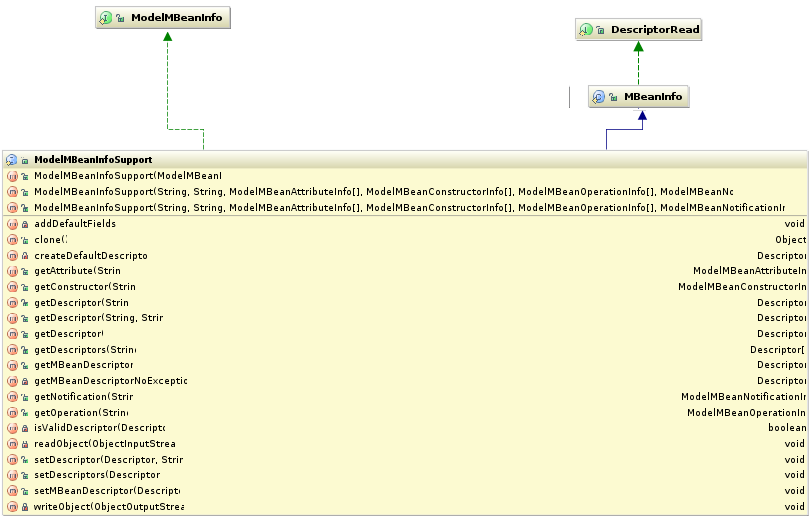
Comme précisé précédemment, un Model MBean est un Dynamic MBean et doit donc implémenter l’interface DynamicMBean.

Il est également à noter que l’implémentation du RequiredModelMBean est dépendante de l’environnement et plus précisément de la JVM : elle peut fournir des mécanismes de persistance, de transaction, de cache, de scalabilité ou de fonctionnement distant en fonction des besoins. Ainsi, le développeur n’a pas à se soucier de problématiques comme celles de persistance en se fiant à l’implémentation de la JVM sur lequel fonctionne son application (par exemple, un environnement J2ME n’a pas besoin d’offrir des propriétés de persistance ou d’accès distant) : il peut le déléguer au Model MBean exposé.

Une implémentation d’un Model MBean doit implémenter l’interface ModelMBean qui étend les interfaces DynamicMBean, PersistentMBean et ModelMBeanNotificationBroadcaster.



En outre, le Model MBean doit exposer ses méta-données dans un objet de type ModelMBeanInfoSupport qui étend la classe MBeanInfo et qui implémente l’interface ModelMBeanInfo.



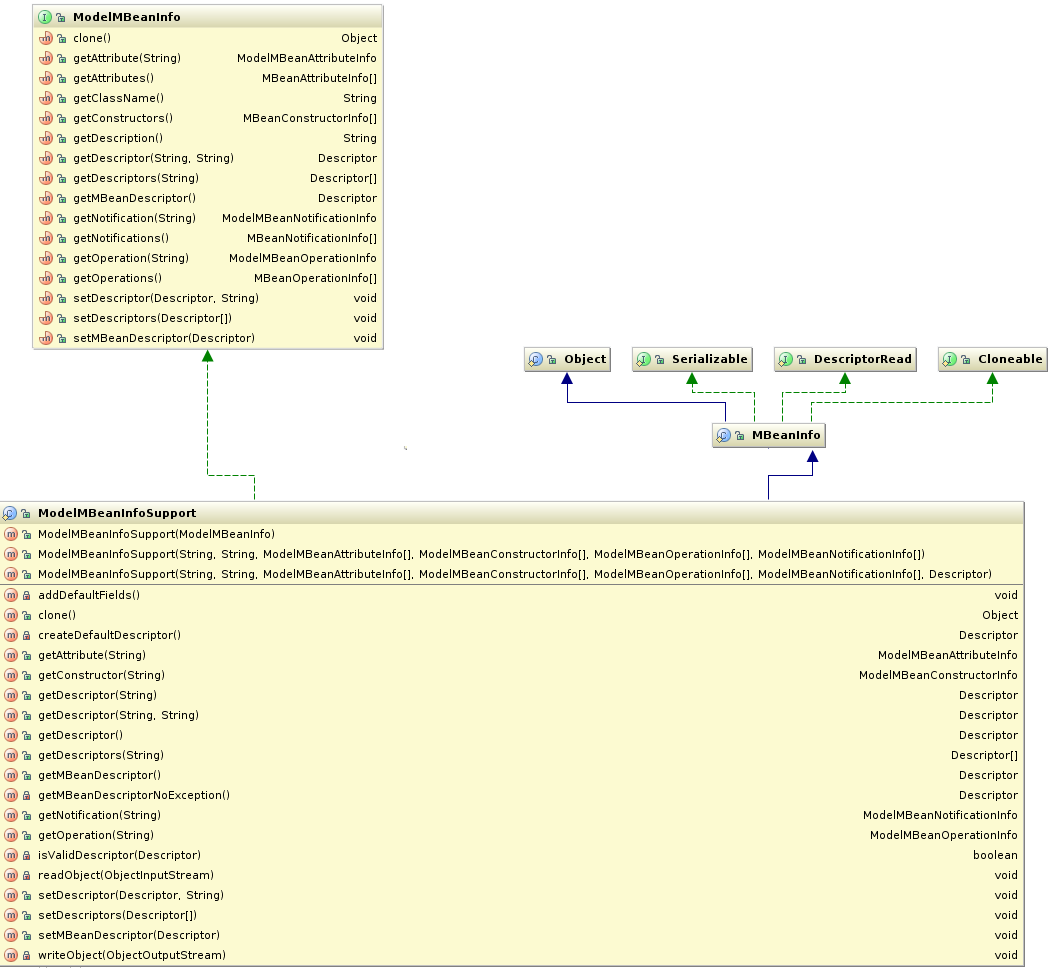
De plus, une instance d’un Model MBean peut émettre des notifications lors de changements de la valeur de ses attributs et possède un constructeur par défaut et un constructeur qui prend en paramètre une instance d’un ModelMBeanInfo. Il est à noter que chaque attribut, constructeur, opération et notification peut fournir sa description. Ces descriptions peuvent être de différentes natures comme la politique de connexion, la politique de notifications, …

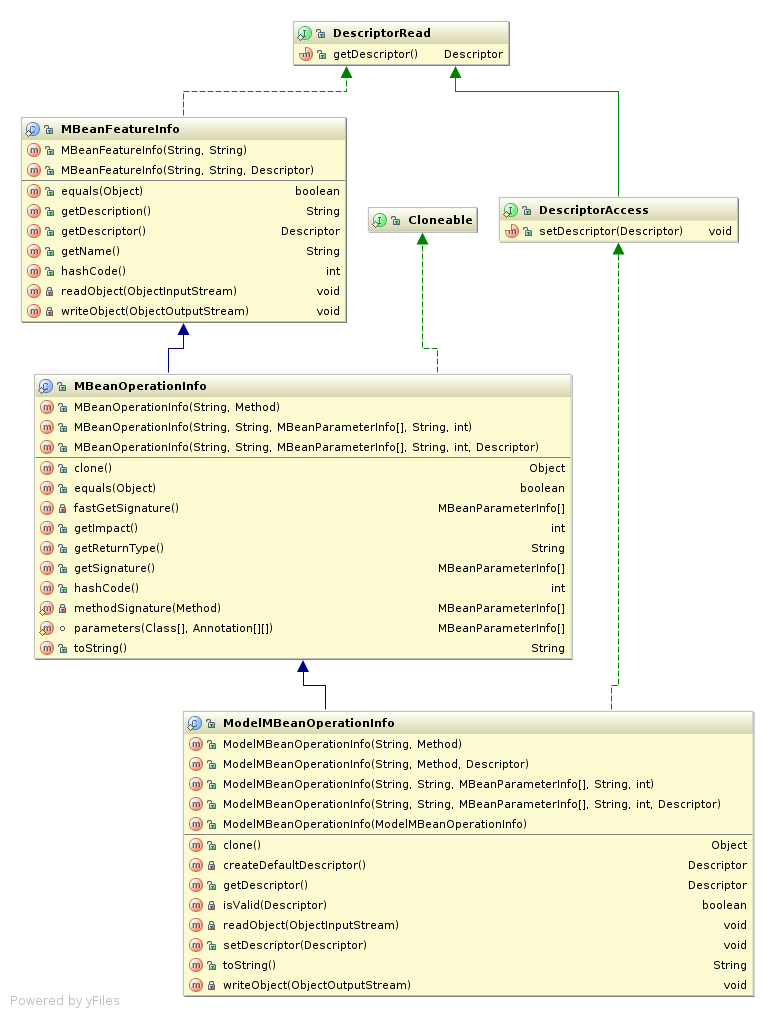
Enfin, les descriptions d’un Model MBean fournissent un mapping entre les attributs et les opérations de l’interface de gestion et les méthodes utilisées par la ressource à gérer (méthodes set, get et invoke). Il est à noter que ces ressources peuvent être indifféremment sur une autre JVM si le Model MBean a été configuré.

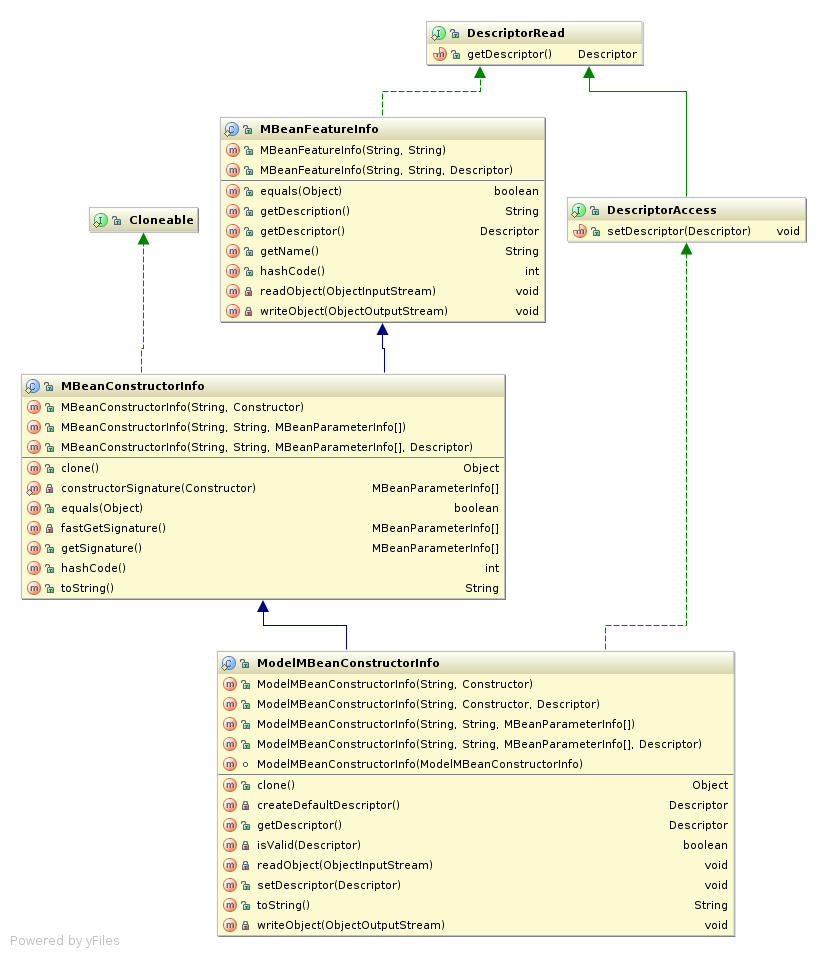
Concrètement, une ressource managée donc doit utiliser l’interface ModelMBeanInfo pour exposer son interface d’administration et de supervision. A l’initialisation, la ressource managée créée ou trouve via le MBean Server une ou des instances d’un ModelMBean qui peuvent alors être configurées avec son interface de management (via une implémentation de l’interface ModelMBeanInfo).

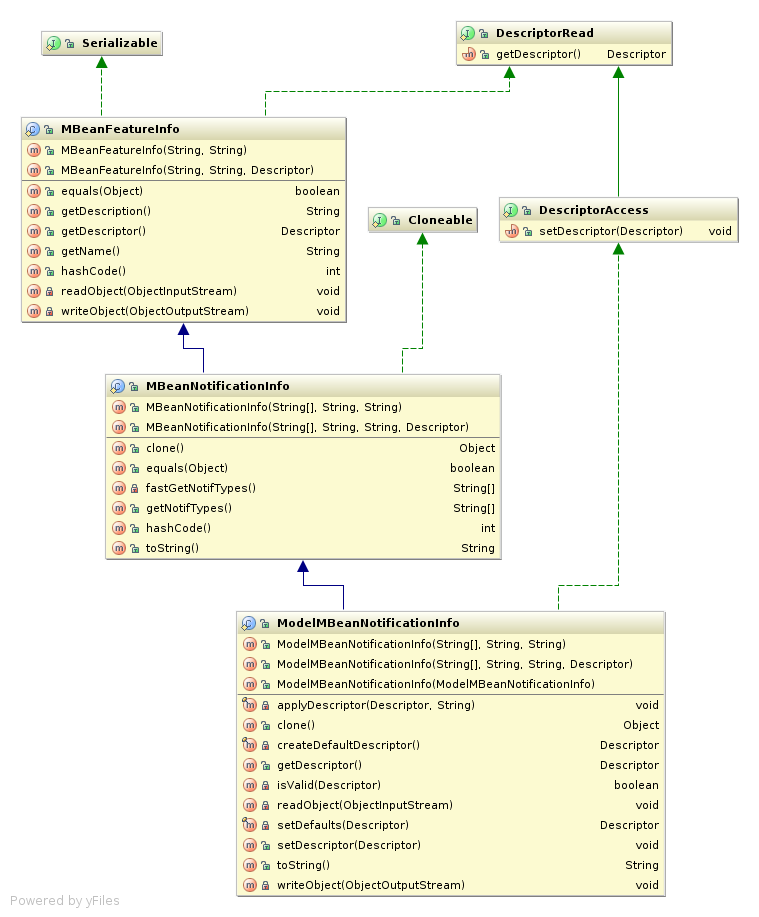
De la même manière, l’application de management (l’application cliente des ressources à administrer et superviser) via le serveur MBean peut accéder aux descriptions, aux opérations et aux notifications du ModelMBean afin de pouvoir les exposer à l’administrateur en lui fournissant le maximum d’information. Afin de permettre au Model MBean de fournir la description de ses différentes opérations, notifications et attributs, JMX propose, par défaut, des sous-classes des classes permettant de décrire un ModelMBean : ModelMBeanInfoSupport, ModelMBeanAttributeInfo, ModelMBeanOperationInfo, ModelMBeanNotificationInfo, ModelMBeanConstructorInfo).

Ces classes implémentent l’interface DescriptorAccess qui permet de remplacer l’interface Descriptor pour chaque attribut, constructeur, opération et notification dans l’interface de management. La description est alors accédée au travers des méta-données de chaque composant.

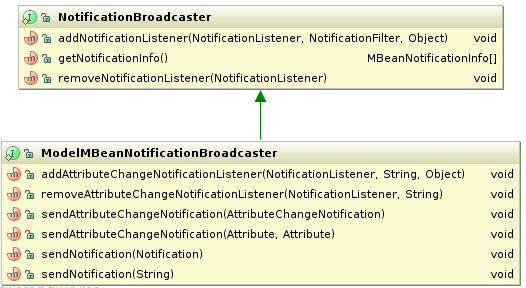






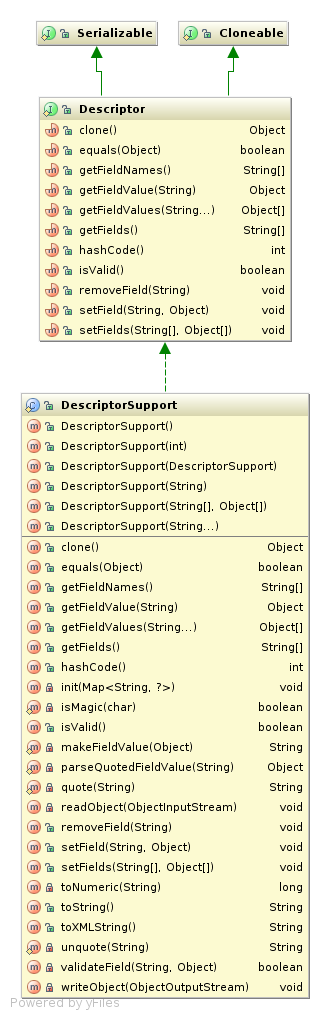


L’interface NotificationBroadcaster, qui permet lorsqu’elle est implémentée par une MBean d’émettre des notifications génériques ou spécifiques et/ou lorsqu’un changement de valeur d’un attribut se produit à des MBean déclarés comme écouteur, est implémentée pour les Model MBean par l’interface ModelMBeanNotificationBroadcaster.



L’interface Descriptor fournit les descriptions des opérations, attributs et notifications du Model MBean qui ne sont pas accédées directement mais au travers de l’interface DescriptorAccess qui définit comment récupérer et positionner la valeur des différents champs qui constituent la description.

Par défaut, JMX fournit deux implémentations de l’interface Descriptor : ImmutableDescriptor et DescriptorSupport.



En fait, l’interface ModelMBeanInfo publie des méta-données sur les attributs, les opérations et les notifications déclarés dans l’interface d’administration et de supervision. Les descripteurs du Model MBean fournissent, quant à eux, le comportement et les règles de ces attributs, opérations et notifications. Un descripteur se présente sous forme d’un ensemble de clé/valeur où la clé est de type String et la valeur est de type Object qui peut être modifié (ajout, modification et suppression des champs) par la ressource managée ou par l’application de gestion. Il est à noter que certains noms sont réservés pour des propriétés prédéfinies telles que la gestion du cache ou de la persistance. Les descripteurs contiennent également le nom des opérations des getter et setter utilisées pour lire et écrire la valeur des attributs.

Comme indiqué précédemment, les descripteurs sont des objets qui implémentent l’interface Descriptor et qui sont accessibles au travers des méthodes définies par l’interface DescriptorAccess dont les méthodes doivent être définies par les implémentations concrètes des interfaces ModelMBeanAttributeInfo, ModelMBeanOperationInfo, ModelMBeanConstructorInfo et ModelMBeanNotificationInfo (qui sont accessibles via l’instance de ModelMBeanInfo).

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Si le descripteur d’un attribut ne possède pas les signatures de ses méthodes, alors l’application ne peut gérer l’attribut : lorsque la méthode setAttribute est invoquée, la valeur est enregistrée dans le descripteur et une notification de type AttributeChangeNotification est émise à destination des écouteurs enregistrés comme intéressés par cet attribut.  Si la méthode getAttribute est invoquée, la valeur est renvoyée par le descripteur mais sa valeur ne peut pas être rafraichie par la ressource managée. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Le Model MBean ne loguera les notifications que si le champ logfile dans le descripteur associé à la notification est renseigné. La journalisation des notifications ne se fera que si l’implémentation du Model MBean ou de l’agent JMX le supporte. Dans le cas contraire, les champs log et logfile seront ignorés. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | La politique de persitance peut être implémentée optionnellement. Cependant, même si elle est géré par le Model MBean, il n’est pas obligé que ce soit ce dernier qui l’implémente et elle peut être offerte par l’agent JMX qui peut proposer en différents niveaux. Lorsqu’aucune politique n’est implémentée, l’objet est transient par défaut. La politique de persistance peut être une implémentation simple offerte en sérialisant dans un fichier les données. Les champs prédéfinis persistLocation et persistName peuvent être utilisés pour définir l’emplacement des données persistées. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | La politique de cache d’une donnée (opération ou attribut) est définie par une valeur par défaut présente dans le descripteur de la donnée. En général, l’adaptateur accède au Model MBean de l’application au travers de l’agent JMX mais une interrogation systématique n’est pas toujours nécessaire (ou voulue) : c’est pourquoi JMX offre la possibitilié de définir un cache d’accès à la ressource. Pour ce faire, la description d’un attribut possède les clés prédéfinies currencyTimeLimit et lastUpdatedTimeStamp. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Le comportement par défaut offert par un Model MBean est généralement suffisant à la plupart des besoins. Cependant, JMX offre la possibilité de relier le Model MBean de l’application à un modèle de données différent. Cela peut se produire dans le cas, par exemple, où un MIB spécifique existe dans le système : le générateur de MIB peut alors intéragir directement avec l’agent JMX et créer un fichier MIB qui sera chargé par un système de supervision SNMP. Pour cela, le descripteur d’un attribut ou d’une opération propose le champ protocolMap qui doit alors contenir une référence à une instance de la classe implémentant l’interface Descriptor. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Si l’implémentation d’un agent JMX supporte les opérations dans un environnement multi-agent JMX, alors il peut avoir besoin d’avertir de son existance et de sa disponibilité un service de recherche. Il peut également avoir besoin d’enregistrer en son sein des MBean qui peuvent être localisés sur d’autres agents JMX sans avoir à connaitre ces derniers. Pour ce faire, le descripteur de son ModelMBeanInfo propose le champ prédéfini export dont la valeur peut être le nom ou l’objet requis pour exporter le MBean. S’il vaut F ou false, alors le MBean ne sera pas exporté. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Il peut parfois être nécessaire d’avoir différent niveau de lisibilité sur les différents MBeans du système. Pour cela, il est possible d’utiliser le champ visibility du descripteur. Il peut prendre une valeur entière de 1 à 4 et peut s’appliquer au niveau du MBean, d’un attribut ou d’une opération. Dans ce cas, c’est à l‘implémentation de l’adaptateur de protocole, du connecteur ou de l’interface de gestion de filtrer les différentes informations à remonter. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Dans le cas où l’interface de gestion doit etre dynamique (ie. qu’elle doit générer son interface en fonction des MBeans découverts ou présents) il est possible d’utiliser le champ presentationString du descripteur. Il doit contenir une chaine de caractères au format XML. Il est à noter, qu’actuellement, il n’existe pas de standardisation de la représentation de ce champ. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Une liste exhaustive des champs prédéfinis des descripteurs peut être trouvée dans la version agrégée de la spécification [JMX au chapitre 4.5 Predefined Descriptor Fields](http://download.oracle.com/javase/6/docs/technotes/guides/jmx/JMX_1_4_specification.pdf). |

Le mot de la fin de cette partie

Ainsi s’achève cette partie qui a permis de voir les notions de MBeans (en particulier les différences entre les différents MBeans) et de Notification.

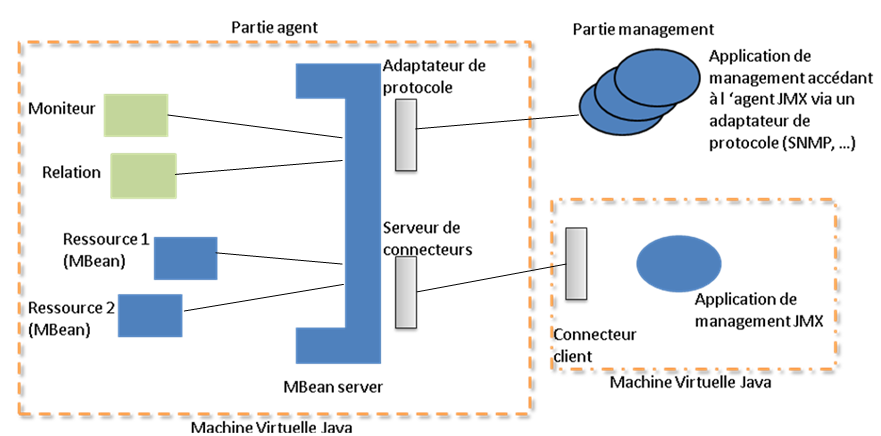
Agent JMX

Un agent JMX est une entité de management qui s’exécute sur une machine virtuelle Java et qui permet de faire la liaison entre les MBean et l’application de management. Un agent JMX est composé d’un MBean server, d’un ensemble de MBean représentant les ressources à administrer et superviser, d’un ensemble minimal de services d’agent qui se présentent sous forme de MBean et d’au moins un adaptateur de protocol ou connecteur.

L’architecture d’un agent JMX peut donc se définir comme tel :

* Les MBeans qui représentent les ressources à administrer et superviser
* Le MBean server qui est la clé de voute de l’architecture d’un agent JMX
* Les services d’agent qui peuvent être des composants pré-définis ou des services spécifiques et qui peuvent être classés de la manière suivante :
  + **Services d’agent dynamiques** qui permettent à l’agent d’instancier des MBeans qui utilisent indifféremment des classes Java ou des librairies natives qui peuvent être téléchargées dynamiquement sur le réseau
  + **Services d’agent de management** utilisés pour superviser la valeur des attributs des MBean : le service notifie sous certaines conditions ses écouteurs lors du changement de valeur d’un attribut du MBean
  + **Services timer d’agent** qui permet d’émettre périodiquement des notifications
  + **Services de relation d’agent** qui définit les associations entre les MBeans et qui est garant de la consistence des relations

Les applications de management peuvent accéder à l’agent JMX aux travers de différents adaptateurs de protocole et connecteurs. Ces objets font parti intégrantes de l’application agent mais ne sont pas définis par les spécifications JMX.



L’architecture JMX permet aux objets d’invoquer les opérations suivantes sur un agent JMX :

* Gérer les MBeans existants en :
  + Récupérant la valeur de ses attributs
  + Changeant la valeur de ses attributs
  + Invoquant des opérations sur ce dernier
* Récupérer les notifications émises par n’importe quel MBean
* Instancier et enregistrer des MBeans à partir de :
  + Classes Java déjà présentent dans la JVM de l’agent JMX
  + Classes Java téléchargées d’une autre machine ou présentent sur le réseau
* Utiliser les services d’agent pour implémenter une stratégie de management qui impactera les MBean existants.

Il est à noter que ces objets peuvent être indifférement du coté de l’agent ou dans une application de management distante.

Les adaptateurs de protocole et les connecteurs rendent l’agent accessible d’une application de management distante et offrent une vue, au travers d’un protocole spécifique, des MBeans instanciés et enregistrés dans le MBean Server.

Les connecteurs sont utilisés pour connecter un agent JMX à une application de management qui supporte JMX, c’est-à-dire une application développée en utilisant des services distribués tels que définis par la spécification JMX. Ce type de communication implique une couche serveur dans l’agent et une couche cliente dans l’application de management.

Un connecteur est spécifique à un protocole donné mais l’application de management peut posséder plusieurs connecteurs indifféremment puisqu’ils ont la même interface.

Un adaptateur de protocole fournit une interface de management de l’agent JMX au travers un protocole donné. Il adapte les opérations du MBean et du MBean Server en une représentation de ce protocole éventuellement avec un modèle différent des informations (comme dans le cas de SNMP). Dans ce cas, l’application de gestion est généralement spécifique au protocole utilisé : elles accèdent à l’agent JMX non à travers une représentation de MBean Server distant mais au travers des opérations qui s’interfacent à celles offertes par le MBean Server. Les serveurs de connecteurs et les adaptateurs de protocole utilisent les services du MBean Server pour invoquer les opérations de l’application de management aux MBeans et pour transmettre les notifications des MBeans aux applications.

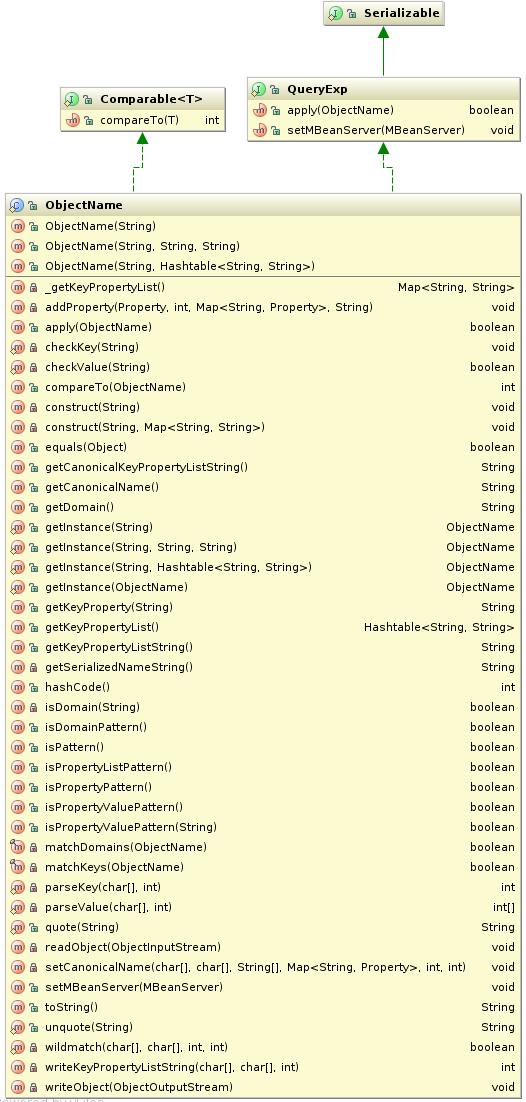
Il est à noter que pour qu’un agent soit gérable, il doit inclure au moins un adaptateur de protocole ou un serveur de connecteurs. Enfin, ces connecteurs et adaptateurs doivent être implémentés comme des MBeans afin qu’ils puissent également être gérable et plus particulièrement afin qu’ils puissent être chargés ou déchargés dynamiquement lorsque cela est nécessaire.

ObjectName

Un nom d’objet permet d’identifier un MBean dans le MBean Server de manière unique. L’application de supervision et d’administration utilise ce nom unique pour identifier le MBean cible.

La classe ObjectName représente ce nom unique dans le MBean Server et est constitué :

* d’un nom de domaine,
* d’un ensemble non ordonné d’une ou plusieurs clés-propriétés.



Le nom de domaine est une chaîne de caractères sensible à la casse qui fournit la structure d’un espace de nommage dans l’agent JMX. Il peut être optionnel car le MBean Server est capable d’en fournir un par défaut.

Il peut contenir n’importe quel caractère mis à part les caractères “:”, “\*” et “?” et prend généralement la forme d’un nom de domaine DNS inversé (ex : com.jetoile.myDomain). Il est déconseillé qu’il contienne la chaine “//”.

La liste de clés-propriétés permet de fournir un nom unique au MBean dans un domain donné. Une clé-propriété est, en fait, un pair de clé/valeur où la valeur est une chaîne de caractères qui ne doit pas contenir les caractères suivants : “:”, “”“, ” “, ”=“, ”\*“, ”?“. Une propriété doit obligatoirement être présente.

Un objectName aura donc la forme suivante :

|  |  |
| --- | --- |
| 1 | [domainName]:property=value[,property=value]\* |

Exemple :

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | MyDomain:description=Printer,type=laser  MyDomain:description=Disk,capacity=2  DefaultDomain:description=Disk,capacity=1  DefaultDomain:description=Printer,type=ink  DefaultDomain:description=Printer,type=laser,date=1993  Socrates:description=Printer,type=laser,date=1993 |

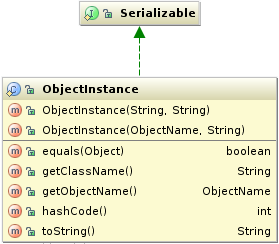
En outre, JMX propose un moyen pour interroger l’annuaire de MBeans présent dans le MBean Server.

Ainsi, pour l’exemple donné ci-dessus, il est possible d’avoir :

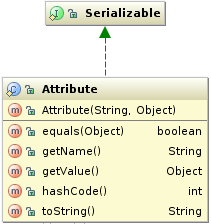
* “*:*” renverra tous les objets du MBean Server. Un objet null ou une chaîne de caractère est équivalente à *.*
* “:\*” renverra tous les objets du MBean Server présents dans le domaine par défaut
* “MyDomain:\*” ne renverra rien
* “??Dom*:*” renverra tous les objets présent dans MyDomain
* “*:description=Printer,type=laser,*” renverra les objets suivants :
  + MyDomain:description=Printer,type=laser
  + DefaultDomain:description=Printer,type=laser,date=1993
  + Socrates:description=Printer,type=laser,date=1993
* “*Domain:description=Printer,*” renverra les objets suivants :
  + MyDomain:description=Printer,type=laser
  + DefaultDomain:description=Printer,type=ink
  + DefaultDomain:description=Printer,type=laser,date=1993
* “*Domain:description=P*,\*” renverra les mêmes résultats que la requête précédente.

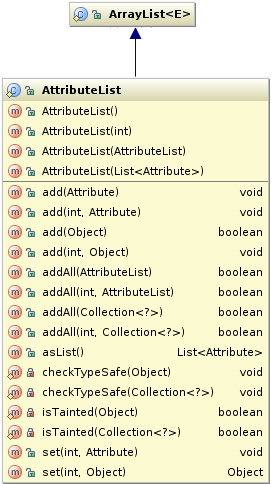
ObjectInstance

La classe ObjectInstance permet de lier le nom d’un objet MBean à sa classe Java. C’est la seule description possible d’un MBean dans le MBean Server puisque l’accès à un MBean par sa référence n’est pas autorisé.



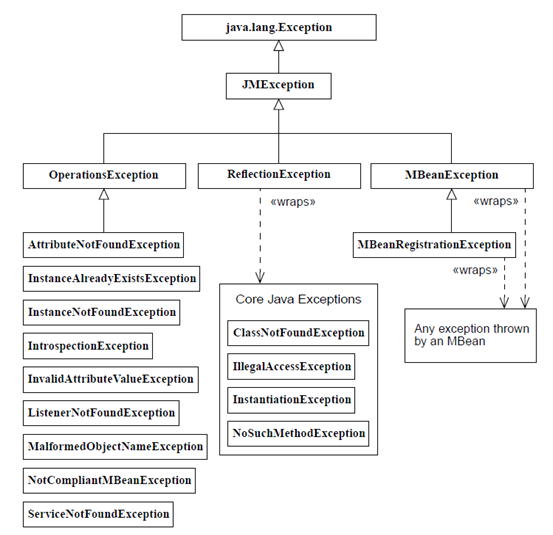
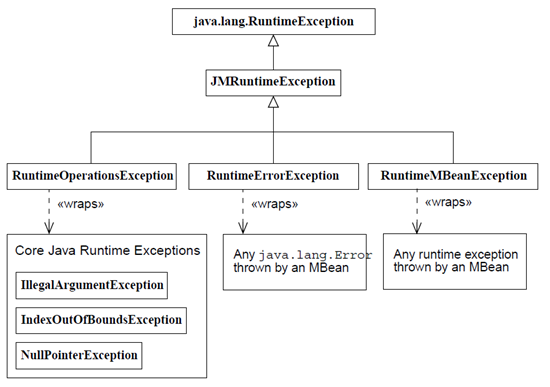
Attribute et AttributeList

Les classes Attribute et AttributeList représente les attributs et leurs valeurs d’un MBean. Elles contiennent le nom des attributs sous forme de chaîne de caractères et leurs valeurs sous forme d’objet de type ObjectInstance.



Les exceptions

JMX propose un ensemble d’exceptions qui peuvent principalement être levées par le MBean Server ou les services de l’agent JMX qui effectuent les opérations sur le MBean lorsque le code de ce dernier lève une exception.



Le mot de la fin de cette partie

Nous avons vu dans cette partie les objets de base manipulés par JMX. Dans les parties suivantes, nous verrons plus précisément la notion de MBean Server.

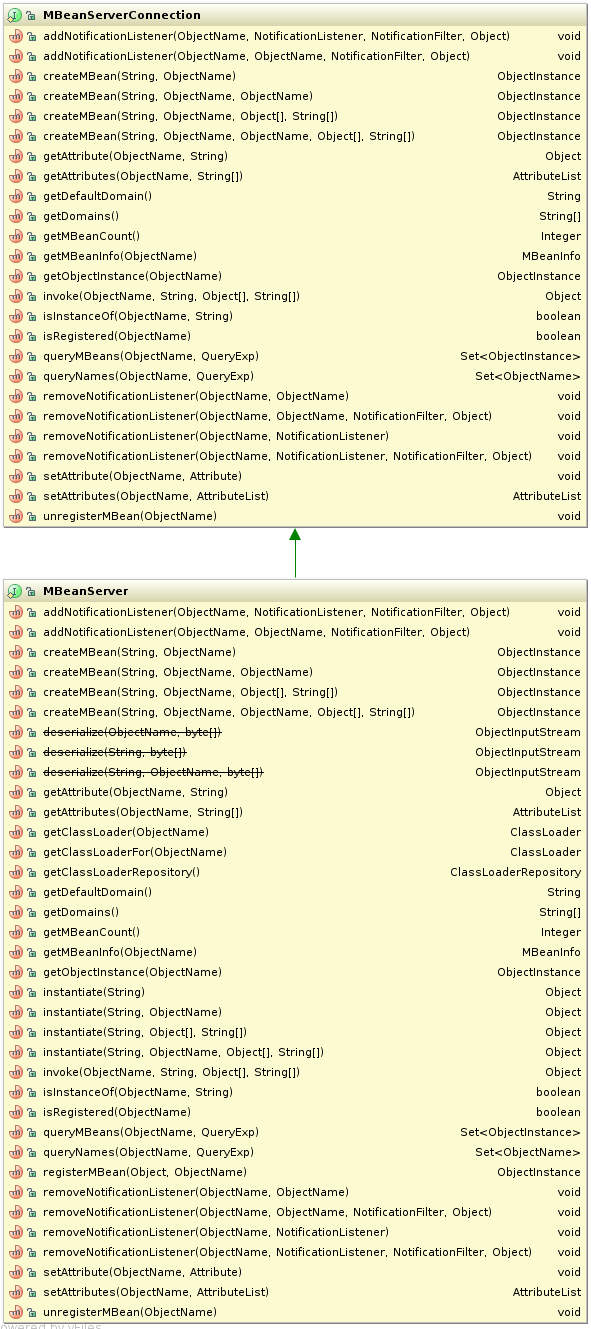
MBean Server

Le MBean Server est un annuaire de MBeans se trouvant dans l’agent JMX. Il permet de fournir les services que manipulent les MBeans et toutes les opérations d’administration et de supervision, qui sont fournies par le MBean, le sont au travers de l’interface MBeanServer :

* Les MBeans qui représentent une ressource managée qui peuvent être n’importe quelle application, des ressources système ou réseaux tant qu’elles sont représentés par une interface Java ou un wrappeur Java.
* Les MBeans qui ajoutent des fonctionnalités de management qui peuvent être génériques (par exemple, des fonctionnalités de journalisation ou de monitoring) ou plus spécifique à une technologie ou à un domaine d’application.
* Les composants de l’infrastructure comme les serveurs de connecteurs et les adaptateurs de protocole.

Pour ce faire, l’agent JMX propose une classe factory pour trouver ou créer un MBean Server qui peut ne pas êre unique dans un agent JMX.

L’interface MBeanServer définie les opérations accessible sur un agent JMX.

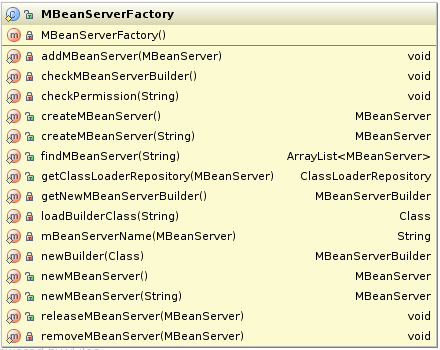


La classe MBeanServerFactory dispose de méthodes statiques permettant de retourner une instance de l’implémentation de l’interface MBeanServer.

Lorsque l’implémentation du MBean Server est créée, il est possible d’indiquer le nom du domaine par défaut utilisé dans l’agent JMX qu’il représente. Ce sont ces méthodes qui sont également utilisées par l’agent JMX pour créer un ou plusieurs MBean Server.

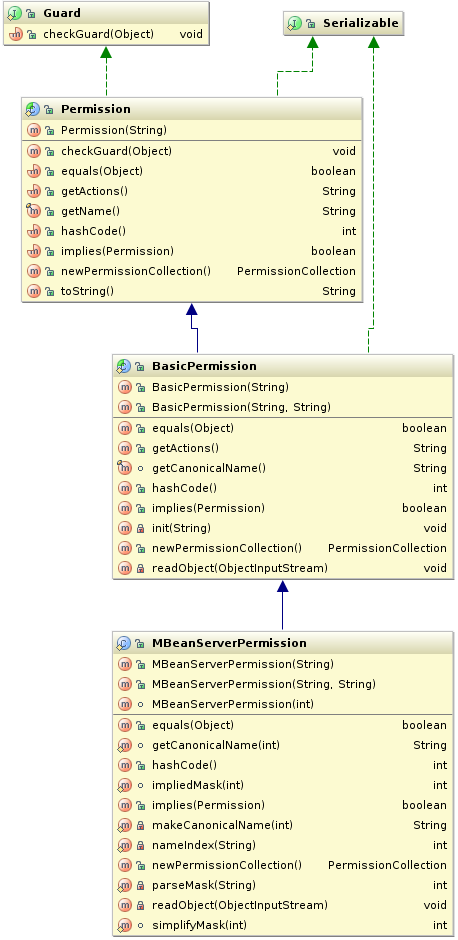
Dans le cas où ce sont des méthodes de recherche et de récupération du MBean Server déjà créé dans l’agent JMX, les classes chargées dans la JVM peuvent accéder aux MBean Server déjà existant sans avoir à connaitre l’agent JMX.

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Depuis Java 5, toute application Java possède un MBean Server qui peut être obtenu via la méthode java.lang.management.ManagementFactory.getPlatformMBeanServer() |



L’accès aux méthodes statiques de la classe MBeanServerFactory est controlé par la classe MBeanServerPermission. Cette classe étend les classes de permissions basiques de Java et permet l’accès aux opérations suivantes :

* createMBeanServer
* findMBeanServer
* newMBeanServer
* releaseMBeanServer



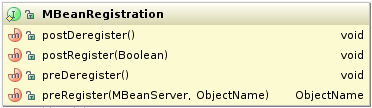
L’enregistrement des MBeans au sein du MBean Server peut être fait soit via l’agent JMX, soit via un autre MBean. Pour ce faire, l’interface MBeanServer propose deux types d’enregistrement :

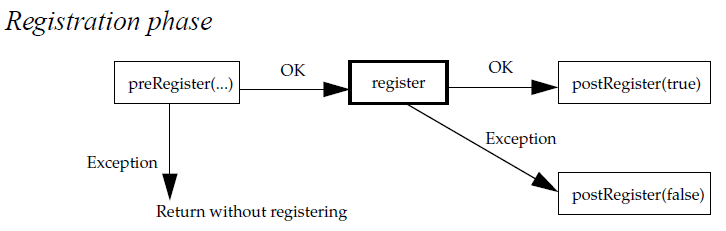
* l’instanciation d’un nouveau MBean et son enregistrement en une seule opération. Dans ce cas, le chargement de la classe Java du MBean peut être fait en utilisant le classe loader par défaut ou en en explicitant un,
* l’enregistrement d’un MBean déjà existant.

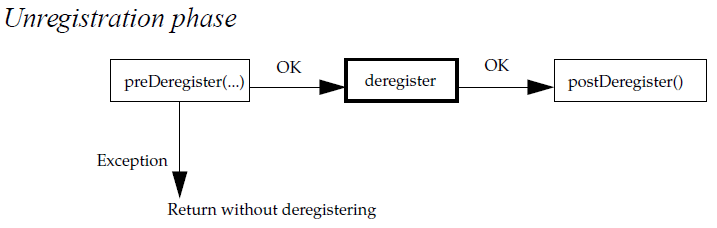
Dans ces deux cas, un ObjectName doit être associé au MBean lors de son enregistrement afin de l’identifier de manière unique au sein du contexte du MBean Server.

Le développeur du MBean peut, de plus, avoir la main sur l’enregistrement et le désenregistrement du MBean au sein du MBean Server en implémentant l’interface MBeanRegistration. Pour ce faire, le MBean Server vérifie dynamiquement avant (resp. après) l’exécution de ces opérations si le MBean implémente l’interface MBeanRegistration. Si c’est le cas, alors les opérations de callbacks adéquates sont invoquées.

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | L’implémentation de l’interface MBeanRegistration est la seule manière de renseigner le MBean sur son environnement et plus particulièrement de lui fournir des informations sur le MBean Server au sein duquel il est enregistré. Cela lui permet, entre autre, d’obtenir des informations sur les autres MBean présent dans le MBean Server. |







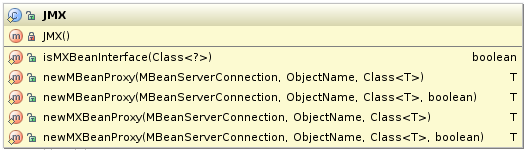
Ainsi, l’interface MBeanServer offre des opérations pour :

* récupérer un MBean à partir de son ObjectName,
* récupérer une collection de MBeans à partir d’une expression régulière sur son nom en appliquant éventuellement un filtre sur ses attributs,
* récupérer la valeur d’un ou plusieurs attributs d’un MBean,
* invoquer une opération sur un MBean,
* d’introspecter un MBean pour récupérer l’interface de management d’un MBean, c’est-à-dire ses attributs et ses opérations,
* s’enregistrer afin de recevoir les notifications émises par un MBean.

Il est à noter que ces méthodes sont génériques, c’est pourquoi elles se basent toutes sur l’objectName du MBean. Le rôle du MBean Server est donc, entre autre :

* de résoudre l’objectName,
* de déterminer si l’opération invoquer sur le MBean peut être consistante (ie. résolue par le MBean),
* de renvoyer le résultat de l’opération au demandeur si les deux points précédents ont été vérifiés.

Cependant, il est également possible, plutôt que d’utiliser l’interface MBeanServer, d’utiliser un proxy. Un proxy est un objet Java implémentant la même interface que le MBean et dont une méthode de son interface est routé au travers du MBean Server vers le MBean. Pour ce faire, la classe statique JMX offre les méthodes utilitaires.



|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Le MBean Server défini un domaine particulier appelé JMXImplementation dans lequel un MBean de la classe MBeanServerDelegate est enregistré. Cet objet identifie et décrit le MBean Server dans lequel il est enregistré. Il est également utilisé pour émettre les notifications émises par le MBean Server : ce MBean agit comme un delegate du MBean Server qu’il représente. Son ObjectName est JMXImplementation:type=MBeanServerDelegate et il possède que des attributs en lecture-seule.  medium |

Les méthodes queryMBeans(ObjectName, QueryExp) et queryNames(ObjectName, QueryExp) de récupération d’un MBean permettent de fournir une requête afin de minimiser la recherche au sein du MBean Server. Pour ce faire, ces méthodes prennent en paramètre :

* un ObjectName sous forme de pattern qui correspond à la portée de la requête et qui permet de spécifier le nom de l’ObjectName des MBeans à rechercher,
* un critère de filtre sur la valeur des attributs qui s’applique aux MBeans trouvés par le premier paramètre.

Dans le cas où la requête ne trouve pas de MBeans ou qu’aucun MBean trouvé ne répond aux critères fournis, le résultat ne contiendra pas d’éléments.

Dans le cas où le pattern de l’ObjectName est null, alors la portée de la recherche se fera dans tout le MBean Server.

Dans le cas où le critère de filtre est null, alors le résultat n’est pas filtré et correspond à tous les éléments se trouvant dans la portée de la requête.

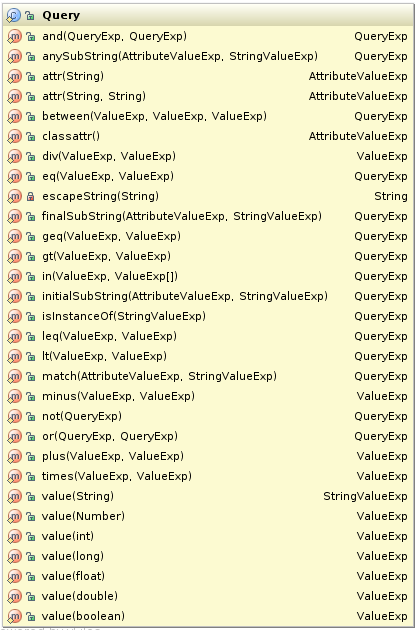
|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Le MBeanServerDelegate est accessible lors de la recherche de MBeans. |

La portée de la requête doit avoir la forme telle que celle vu dans la [quatrième partie](https://blog.jetoile.fr/2010/11/jmx-pour-les-nuls-les-classes-de-base.html) de cette série de post sur JMX.

Le critère de filtre doit utiliser les classes et interfaces suivantes :

* L’interface QueryExp qui permet d’identifier les objets qui forme les expressions de la requête. Ces objets peuvent être utilisés dans une requête ou peuvent composés une requête plus complexe.
* Les interfaces ValueExp et StringValueExp qui permettent d’identifier les objets qui représentent les valeurs numériques ou de type chaîne de caractères.
* L’interface AttributeValueExp qui permet d’identifier les objets qui représentent les attributs.
* La classe Query qui permet la construction d’une requête. Elle propose des méthodes statiques qui retournent des objets de type QueryExp et ValueExp.
* La classe ObjectName qui implémente l’interface QueryExp et qui peut donc être utilisé dans un critère de filtre.

Dans la pratique, il n’est pas nécessaire d’avoir à utiliser directement les interfaces ValueExp et QueryExp. En effet, la classe Query fonctionne comme une fabrique d’objets de type QueryExp.



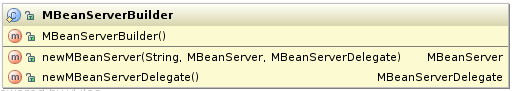
Ainsi, on pourra, par exemple, avoir le code suivant :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | QueryExp exp = Query.and(  Query.geq(Query.attr("age"),  Query.value(20)),  Query.match(Query.attr("name"),  Query.value("G\*ling")));  QueryExp exp = Query.eq(  Query.classattr(),  Query.value("managed.device.Printer")); |

L’interface MBeanServerConnection étendue par l’interface MBeanServer permet de fournir une interface commune d’accès au MBean Server qu’il soit distant ou local.

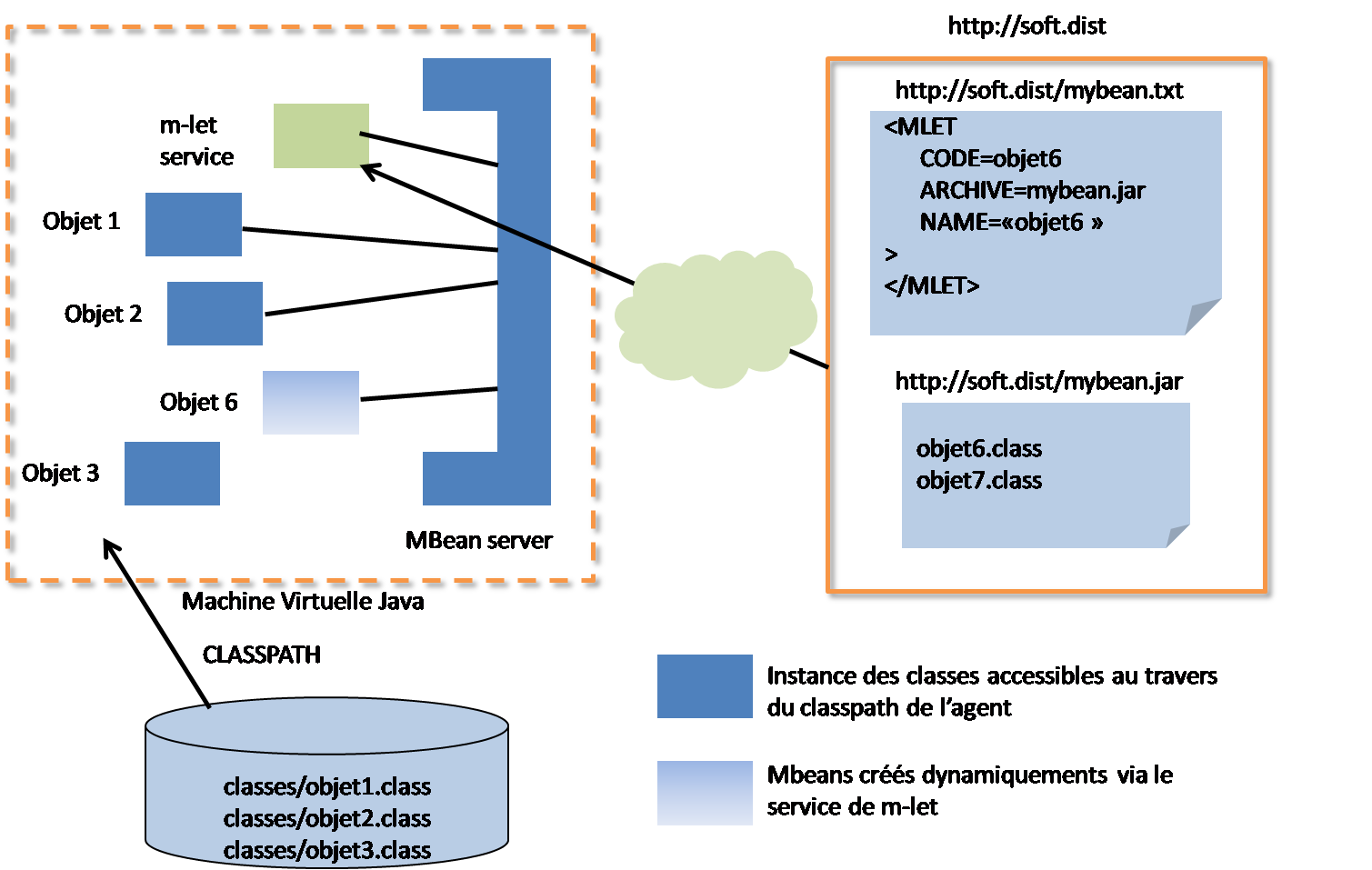
Enfin, il est possible de fournir sa propre implémentation du MBean Server en renseignant la propriété système javax.management.builder.initial : lorsque les méthodes createMBeanServer() ou newMBeanServer() sur les MBeanServerFactory sont appelées, alors, si la variable système est renseignée, c’est le MBean Server qui est renseigné qui est utilisé, sinon, c’est le MBean Server par défaut.

Ce MBean Server doit être une classe publique qui étend la classe `MBeanServerBuilder. Il est à noter que cette classe doit également pouvoir créer une instance de MBeanServerDelegate (qui peut être standard ou pas).

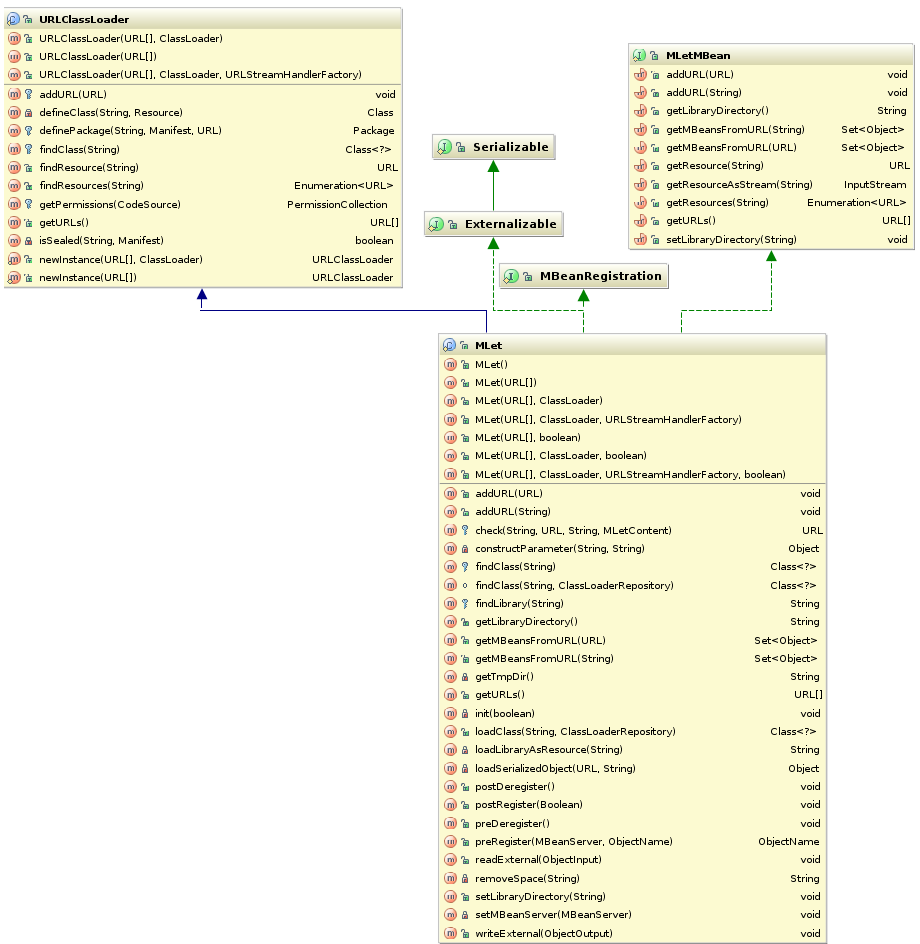


Chargement dynamique des MBeans

JMX propose un service de chargement dynamique de MBeans qui s’appuie sur les fonctionnalités du classloader Java pour récupérer et instancier les MBeans qui peuvent utilisés indifféremment des classes Java ou des librairies natives. Cette fonctionnalité existe car lorsque le serveur JMX est démarré, il peut ne pas avoir connaissance de tous les MBeans et devoir en télécharger. Pour ce faire, un service particulier chargé de gérer des applets de management (m-let) est utilisé pour instancier les MBeans obtenus d’une URL distante sur le réseau. En fait, le service de m-let permet d’instancier et d’enregistrer des MBeans dans le MBean Server en chargeant un fichier texte, dont la localisation est spécifiée par une URL, qui contient les informations des MBeans à obtenir. Quand ce fichier texte est chargé, toutes les classes spécifiés par le tag MLET sont téléchargées et une instance de chaque MBean spécifié est crée et enregistré. Le service de m-let étant lui-même un MBean enregistré dans le MBean Server, il peut être utilisé par d’autres MBeans, par d’autres applications ou par d’autres applications d’administration et de supervision distantes.



|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Depuis Java 5, toute application Java possède un MBean Server qui peut être obtenu via la méthode java.lang.management.ManagementFactory.getPlatformMBeanServer() |



|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | La classe MLet étend la classe URLClassLoader, ce qui implique qu’il s’agit également d’un classe loader. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Dans le cas générique, l’appel à la méthode *getMBeansFromURL()* du service de m-let permet de :   * télécharger le fichier texte contenant le descriptif des MBeans (un MBean par tag MLET), * télécharger les classes nécessaires, * charger les classes dans le classe loader du MBean Server, * instancier les MBeans, * enregistrer les MBeans instanciés préalablement, * renvoyer les instances des MBeans. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Dans le cas où le MBean contient de méthodes natives et que des librairies natives sont chargées en utilisant la méthode *System.loadLibrary*, alors la méthode *findLibrary* du classe loader du m-let est appelé pour trouver la librairie. Cette méthode surcharge celle de la classe *ClassLoader* et, si la librairie est trouvée, copie cette dernière à l’emplacement retourné par la méthode *getLibraryDirectory* de la classe *MLet*. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Le MBean Server maintient une liste de classe loader dans le repository de classe loader (aussi appelé *default loader Repository*).  Ce classe loader *Repository* est utilisé dans les méthodes createMBean et instantiate de l’interface MBean Server : le repository loader repository est utilisé pour trouver et charger les classes. Si un m-let ne trouve pas les classe à l’URL, il essaie de chargeer les classes du classe loader repository. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Dans le cas où plusieurs MBean Server sont présent dans une même JVM, ils disposent chacun de leurs propres classe loader repository qui sont indépendants les uns des autres. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | L’ordre des classes loader dans le repository est important : quand une classe est chargée dans le repository, chaque classe loader est interrogé pour charger la classe. Si un loader réussi à charger la classe alors la recherche s’arrête, sinon (s’il lève une exception *ClassNotFoundException*) la recherche continue jusqu’à ce qu’il n’y en ait plus (dans ce cas l’exception *ClassNotFoundException* est levée). Le premier loader dans la classe loader repository est celui qui a chargé l’implémentation du MBean Server. Se trouve ensuite les loaders des MBeans, l’ordre des loaders étant celui du chargement des MBeans. Ainsi, on aura, par exemple :  Un MBean m1 apparait avant le MBean m2 si la méthode *createMBean* ou *registerMBean* qui a enregistrée m1 s’est terminé avant le démarrage de l’opération qui a enregistré m2. Si la méthode d’enregistrement de m2 a démarré avant la fin de la méthode d’enregistrement de m1, alors l’ordre des loaders est indéterminé. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Un m-let est un classe loader et, comme l’indique le comportement d’un classe loader standard, il charge une classe en utilisant la méthode *loadClass* : En premier lieu, il envoie une requête au classe loader parent pour charger la classe (le classe loader parent est celui spécifié lors le m-let est créé (par défaut le classe loader système)), puis, dans un second temps, si le classe loader parent n’a pas pu charger la classe, le m-let tente de la charger dans sa liste d’URL. Enfin, si ces deux essais se soldent par un échec, le m-let tente de charger la classe dans le classe loader repository (dans ce cas, il est dit qu’il délègue au classe loader repository). Si ces trois tentatives échouent, l’exception *ClassNotFoundException* est levée.  Pour avoir d’autres informations sur le chargement de classe ainsi que sur les problèmes qui peuvent apparaitre, voir les chapitres *8.4 The Class Loader Repository* et *8.5 Using the Correct Class Loader for Parameters* des spécifications JMX. |

Service Monitoring

Il existe une famille de MBeans de monitoring qui se présentent qui permettent de scrupter les variations au cours du temps de la valeur des attributs des autres MBeans et qui permet d’émettre des notifications à intervalle régulier. Ils sont aussi classifiés comme des services de monitoring.

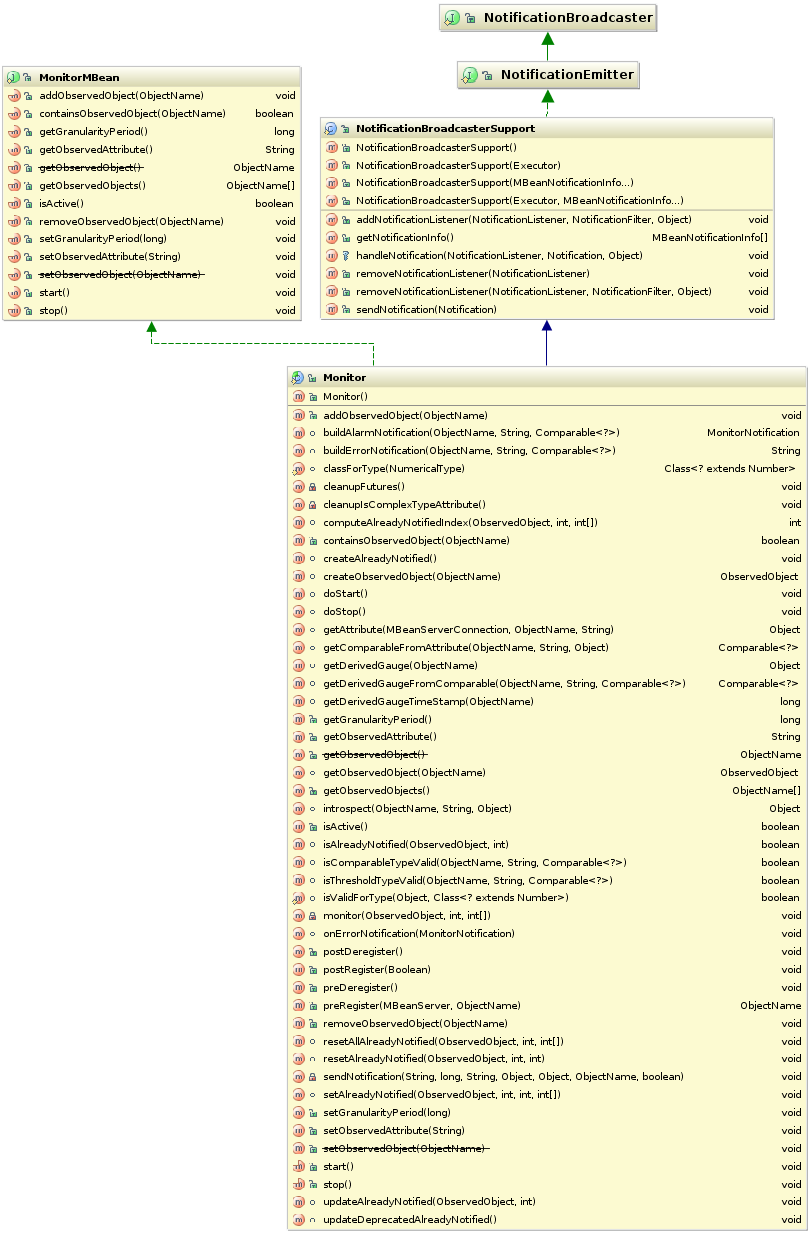
En fait, un service de monitoring (et donc un MBean de monitoring) permet d’observer la valeur d’un attribut donné présent dans un ou plusieurs MBeans (on dit qu’il s’agit alors du **MBean observé**) à un intervalle à régulier. Cette valeur peut être indifféremment la valeur d’un attribut ou la valeur contenue dans la valeur d’un attribut s’il est de type complexe. Pour chaque MBean observé, le moniteur interpole une seconde valeur qui est appelé le *pas dérivé* (**derived gauge**) et qui correspond, soit à la valeur observée, soit à la différence entre deux valeurs numériques obtenues consécutivement.

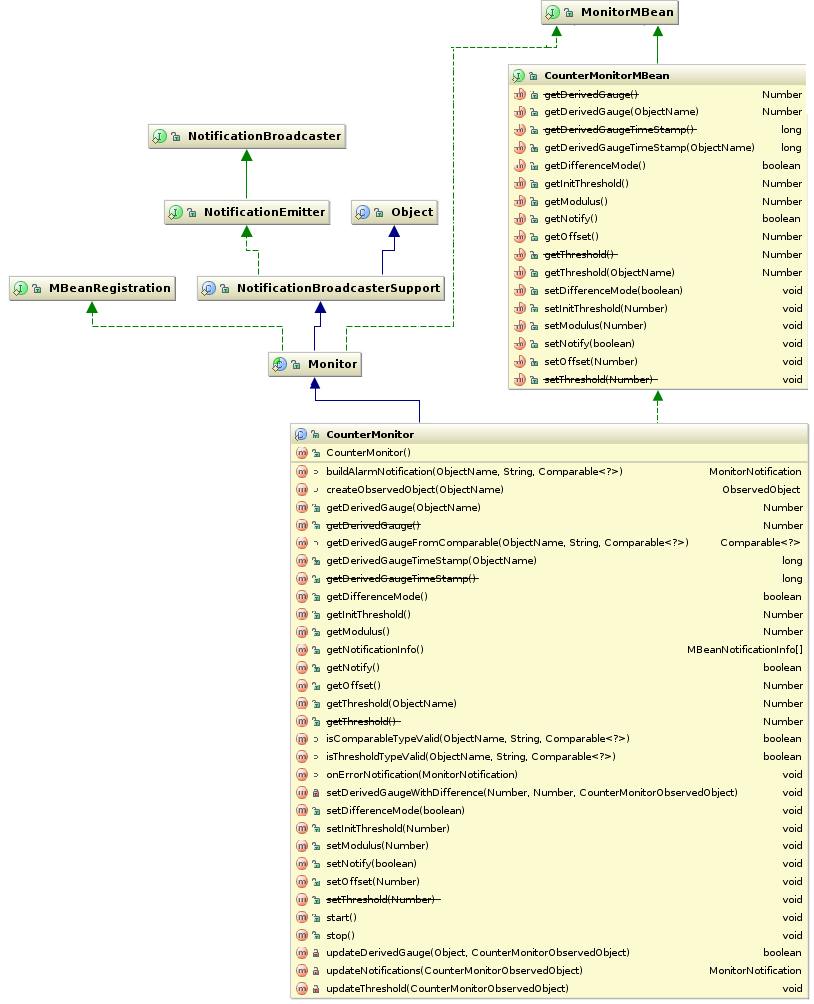
Un type de notification spécifique est alors émis à chaque service de monitoring lorsque le pas dérivé répond à un ou un ensemble de conditions (ou lorsque certaines erreurs se produisent) qui peuvent être initialisées lors de l’initialisation du moniteur ou dynamiquement au travers de l’interface d’administration du MBean moniteur.

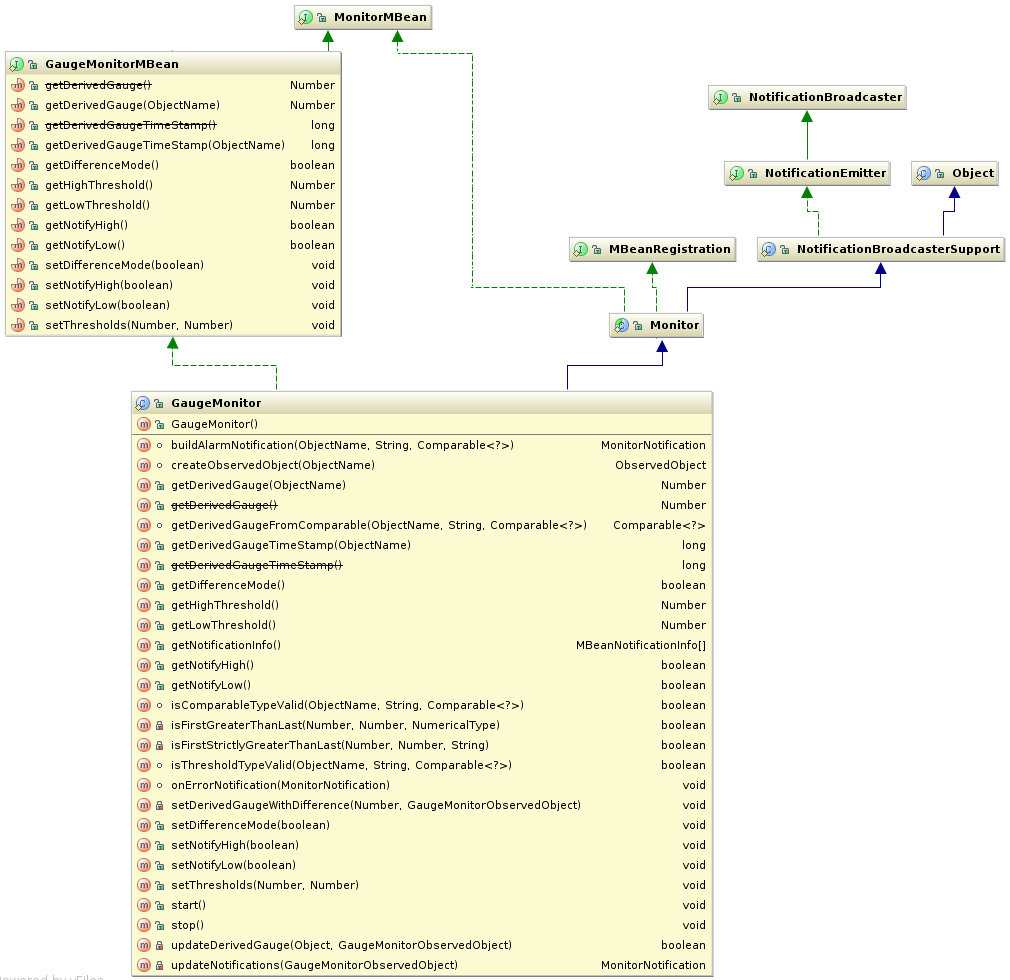
Il existe trois types de moniteurs :

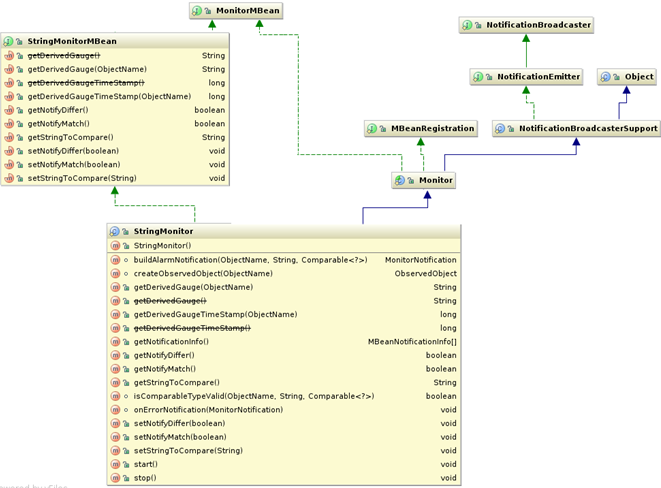
* **CounterMonitor** qui permet de scrupter des valeurs de type entier (Byte, Integer, Short, Long) et qui se comporte comme un compteur :
  + les valeurs ne peuvent être inférieures strictement à 0,
  + les valeurs ne peuvent qu’être incrémentées,
  + les valeurs peuvent être tournantes (dans ce cas un modulo doit être défini).
* **GaugeMonitor** qui permet de scrupter des valeurs de types entier ou flottantes (Float, Double) et qui se comporte comme un pas
* **StringMonitor** qui permet de scrupter des valeurs de type String.

Tous ces moniteurs doivent étendre la classe abstraite Monitor alors que les types des valeurs surveillées doivent être supportés par le type de moniteur utilisé. Cependant, il est à noter que le moniteur vérifie le type de la valeur de l’instance retournée et non le te type de l’attribué déclaré dans les méta-données du MBean observé.





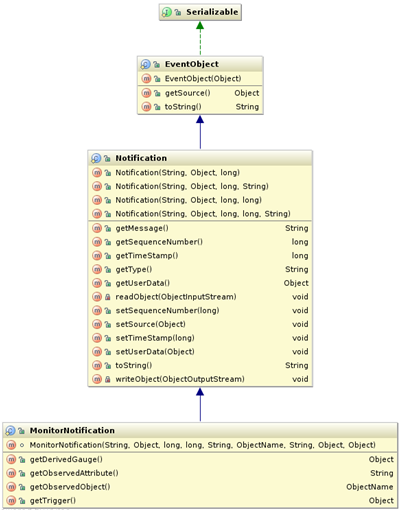




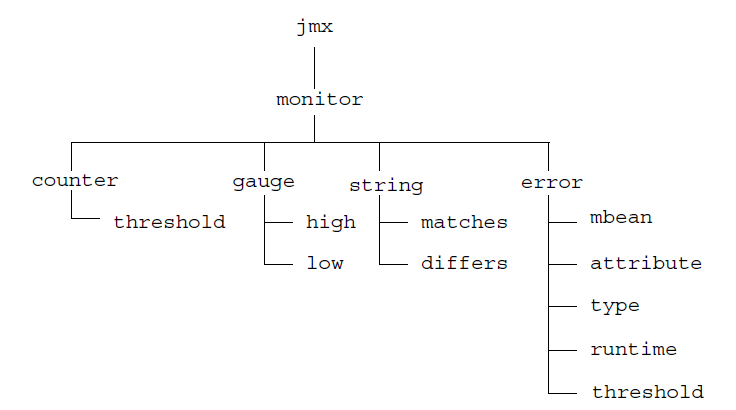
Pour les notifications, une sous classe spécifique de Notification est définie pour être utilisée par tous les services de monitoring. Il s’agit de la classe MonitorNotification.

Ainsi, elle est utilisée pour rapporter les cas suivants :

* Une condition de déclenchement d’un moniteur est détectée (par exemple, si le seuil d’une valeur est atteinte)
* Une erreur s’est produite durant la scrutation d’un attribut (par exemple, si un MBean est désenregistré)



La figure suivante représente tous les types de notifications qui peuvent être générées par un service Monitor.



Service Timer

Le service Timer permet de déclencher des notifications à des dates et heures spécifiques ou à intervalles réguliers. Les notifications sont émises à tous les objets déclarés comme étant interessés par les notifications émises par le timer (patron **Observer**).

Le service Timer est un MBean qui peut être managé en permettant, entre autre, aux applications de modifier la configuration de son ordonnanceur (**scheduler**).

En fait, la classe Timer gère une liste de notifications datées qui sont émises quand leur date et heure arrivent. Les méthodes de cette classe permettent d’ajouter et de supprimer des notifications de sa liste. En fait, les types de notifications sont fournis par l’utilisateur avec la date et optionnellement la fréquence et le nombre de répétitions.

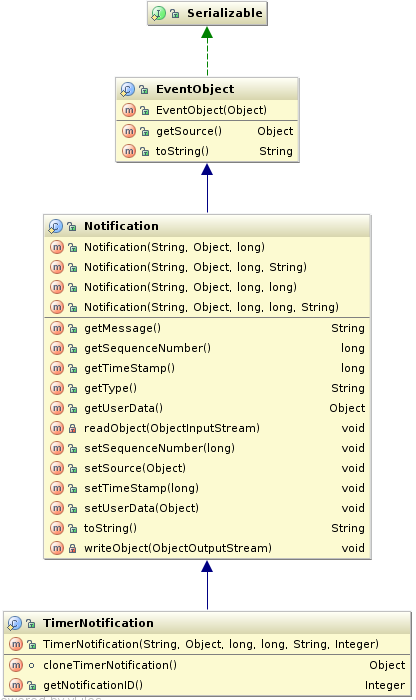


Pour les notifications, une sous classe spécifique de Notification est définie pour être utilisée par le service **Timer**. Il s’agit de la classe TimerNotification.

Ainsi, elle est utilisée pour rapporter les cas suivants :

* Les notifications ne sont déclenchées qu’une seule fois
* Les notifications sont répétées à une période définie et/ou un nombre d’occurrences

Ces paramètres sont définis lorsque la notification est ajoutée à la liste de notifications du timer. En outre, à chacune des notifications émises est associée un identifiant unique.



|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Si la notification ajoutée au service Timer possède une date antérieure à la date courante, alors la méthode addNotification se comporte comme si la date indiquée était la date courante. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Si la date d’émission d’une notification est mise à jour alors qu’elle est déjà existante, alors aucune notification de mise à jour n’est émise. |

Le service Timer MBean est un émetteur standard de notifications (qui possèdent un type et une date définis dans la liste de notifications ajoutées à l’aide de la méthode addNotification()). Tous les écouteurs d’un MBean Timer donnés reçoivent toutes les notifications. Les écouteurs interessés par un type de notifications Timer particulier doivent spécifier un objet filtre lors de leur enregistrement.

Lorsqu’un timer est actif et que la date d’une notification de type Timer arrive à son terme, le service Timer émet cette notification en fournissant son type, message, données utilisateurs ainsi que l’identifiant de la notification. S’il s’agit d’une notification périodique disposant d’un nombre d’occurrence, il est décrémenté.

Enfin, si la notification n’a plus besoin d’être émise (notification non récurente ou notification périodique disposant d’un nombre d’émission prédéfini), elle est supprimée de la liste des notifications.

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Si une notification dispose d’une date d’émission antérieure à son démarrage, il est possible de l’émettre ou de l’ignorer à l’aide de la méthode sendPastNotification(). Par défaut, la valeur de cet attribut est mise à false. |

Service Relation

JMX définit un modèle de relation entre les MBeans. Une relation est définie par l’utilisateur et se présente comme une association n-aire entre les MBeans.

Pour ce faire, JMX propose un ensemble de classes à utiliser pour décrire les relations, ainsi qu’un service qui permet de centraliser toutes les opérations de mise en relation.

Toutes les relations permettent d’associer un type de relations (qui fournit des informations sur les rôles qu’il contient comme la multiplicité) avec le nom de classe du MBean qui satisfait le rôle.

Le service de relations permet également de chercher un MBean au travers de ses relations.

Il est garant de la consistance des relations en vérifiant les opérations et le désenregistrement des MBean afin de s’assurer que les relations sont toujours conformes. Si une relation n’est plus valide, elle est supprimée du service de relations même si ses MBeans membres restent présents.

Une relation est composée de rôles nommés qui disposent d’une liste de MBeans associés à ce rôle. Cette liste doit être consistante avec les informations de multiplicité ainsi qu’avec les classes des MBean qui représentent ce rôle.

Un ensemble de définition de rôles constitue un type de relation.

Un type de relation est un patron pour toutes les instances de relations qui associent les MBeans représentant ses rôles.

Par la suite, le terme de relation signifiera une instance spécique d’une relation associant des MBeans existants aux rôles dans un type de relations défini.

Le tableau ci-dessous décrit les concepts nécessaires à la compréhension du modèle de relations défini par JMX :

| **Concept** | **Description** |
| --- | --- |
| Role information | Décrit un des rôles dans une relation. L’information sur un rôle est composée du nom du rôle, de sa multiplicité, du nom de la classe qui participe à ce rôle, d’une propriété en lecture-seule sur les permissions et d’une description. |
| Relation type | Méta-donnée d’une relation, composée d’un ensemble de Role information. Permet de décrire les rôles qu’une relation satisfait et permet de servir de patron pour créer et maintenir des relations. |
| Relation | Association courante entre des MBeans qui satisfont une relation donnéee. Une relation peut avoir des propriétés et méthodes qui ne s’appliquent qu’à ses MBeans. |
| Role Value | Liste de MBeans qui satisfont un rôle donné dans une relation. |

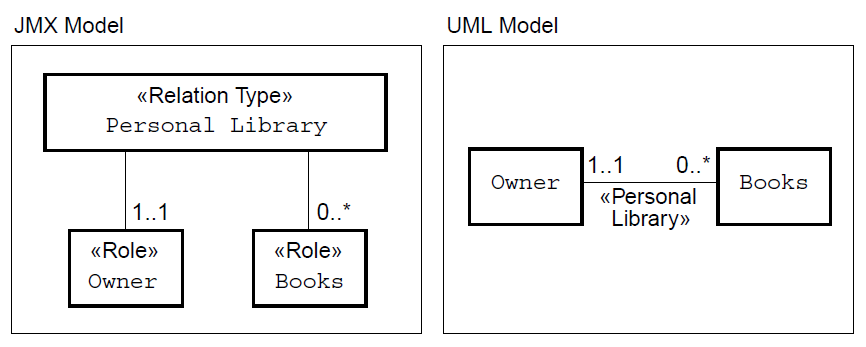
Unresolved Role Résultat d’un accès illégal d’une opération sur un rôle. Contient la raison du refus de l’opération. Support Classes | Classes internes utilisées pour représenter des types de relations et des instances de relations. Relation Service | MBean qui peut accéder et maintenir la consistance de tous les types de relations et toutes les instances de relations dans un agent JMX. Fournit des opérations pour trouver un MBean et ses rôles dans une relation.

En JMX, le type de relation est un ensemble statique de rôles. Les types de relations peuvent être définis à l’exécution mais, une fois défini, ces rôles et informations sur les rôles ne peuvent être modifiés.

Exemple de relations : Dans cet exemple, des livres (*Books*) et les propriétaires (*Owner*) représente des rôles :

* Books représente le nombre de livres que possède la classe d’un MBean donnée.
* Owner représente un propriétaire de livres d’une autre classe d’un MBean.

Un type de relations (*Personal Library*) peut être défini comme étant la notion d’appartenance de livres.



En fait, le service de relations empêche la création de types de relations invalides, de relations invalides ou la modifiation de valeurs de rôles invalides.

Quand une relation est supprimée du service de relations, ses MBeans membres ne sont pas impactés. Quand un type de relation est supprimé, toutes les relations référencées sont au préalable supprimées. Cependant, c’est à l’appelant de gérer la consistance lors de la suppression d’un type de relations.

En outre, puisque les relations ne sont définies qu’au niveau des MBeans enregistrés, en supprimer un de l’annuaire est susceptible de modifier la relation. Le service de relations écoute, pour tous les MBean, le server de notifications qui indique quand un membre d’une relation est désenregistré. Le MBean correspondant est alors supprimé de tous rôles où il se trouve. Dans le cas où la nouvelle cardinalité du rôle n’est pas consistante avec son type de relation, alors la relation est supprimée du service de relation.

Le service de relations émet des notifications après l’exécution d’opérations qui modifient une instance de relations (création, mise à jour ou suppression). Cette notification fournit des informations sur la modification comme l’identifiant de la relation et la nouvelle valeur d’un rôle.

De plus, il est à noter qu’il existe deux types de relations et relations :

* Les types de relations et relations internes : les types de relations et leurs instances sont créés par le service de relations et ne sont accessibles qu’aux travers ses opéraitons.
* Les types de relations et relations externes : les types de relations et leurs instances sont instanciés en dehors du service de relations puis lui sont ajoutés. Dans ce cas, ils peuvent être accessibles directement et, en l’occurrence, peuvent être enregistré comme des MBeans. Il est cependant à noter que ces types de relation sont immuables et qu’elles ne doivent pas être modifiées une fois enregistré dans le service de relation.

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | En raison de la complexité de ce service, pour plus d’informations, il est préférable de se référer aux chapitres 11.2 Relation Service Classes, 11.3 Interfaces and Support Classes et 11.4 Role Description Classes des spécifications agrégées JMX. |

Service securité

Un MBean Server JMX peut avoir accès à des informations sensibles et peut être susceptible d’exposer des opérations sensibles. Pour ce faire, JMX propose un mécanisme d’accès à de telles opérations en s’appuyant sur le modèle de sécurité de Java : il est possible de définir des permissions qui permettent de contrôler l’accès au MBean Server et à ses opérations.

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | La sécurisation du MBean Server s’appuyant sur le modèle standard de sécurité Java, il est nécessaire de disposer du gestionnaire de sécurité Java (security manager). La méthode statique System.getSecurityManager() permet de le vérifier. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Pour rappel, le modèle de sécurité de Java s’appuie sur le principe de permissions : pour qu’une opération soit invocable, l’appelant doit posséder la/les permissions. En effet, à un point donné de l’exécution d’un programme correspond un ensemble de permissions appliqué à l’exécution d’un thread.  En fait, une opération est autorisée si le thread qui l’invoque dispose des permissions nécessaires.  Les permissions sont des objets java qui étendent la classe *Permission*.  [https://1.bp.blogspot.com/_XLL8sJPQ97g/TPKS5FhDvOI/AAAAAAAAAQk/QbB4KTZsUXA/s320/jmx66.png](https://1.bp.blogspot.com/_XLL8sJPQ97g/TPKS5FhDvOI/AAAAAAAAAQk/QbB4KTZsUXA/s1600/jmx66.png) |

En fait, JMX défini, par défaut, sa propre permission MBeanServerPermission qui peut être configurée comme suit :

* MBeanServerPermission qui prend en paramètre de son constructeur une chaine de caractère représentant la liste des opérations autorisées sur le MBeanServer (ex : MBeanServerPermission("createMBeanServer") qui permet de controler l’accès aux méthodes MBeanServerFactory.createMBeanServer )
* MBeanPermission qui prend en paramètre deux chaines de caractères :
  + la première représente la cible et peut être composée du nom canonique de la classe ou d’une expression régulière, du membre (c’est-à-dire le nom de l’attribut ou de l’opération qui doit être accédé), et de l’ObjetName du MBean cible.

Exemples de cible :

|  |  |
| --- | --- |
| 1  2  3 | com.example.Resource#Name[com.example.main:type=resource]  com.example.Resource#\*[com.example.main:type=resource]  \*#Name[com.example.main:type=resource] |

* + la deuxième représente l’action une ou plusieurs méthodes du MBean Server parmi les suivantes :
    - `addNotificationListener`
    - `getAttribute`
    - `getClassLoader`
    - `getClassLoaderFor`
    - `getClassLoaderRepository`
    - `getMBeanInfo`
    - `getObjectInstance`
    - `instantiate`
    - `invoke`
    - `isInstanceOf`
    - `queryMBeans`
    - `queryNames`
    - `registerMBean`
    - `removeNotificationListener`
    - `setAttribute`
    - `unregisterMBean`

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Les opérations suivantes ne sont pas soumises à la sécurisation du MBean Server :   * *isRegistered* * *getMBeanCount* * *getDefaultDomain* |

De plus, JMX propose la permission MBeanTrustPermission qui permet de faire confiance à un certain utilisateur ou code source signé. Si le signataire ou le code source possède cette permission, le MBean est considéré comme étant de confiance. Seuls les MBeans de confiance peuvent alors être enregistrés dans le MBean Server.

Une permission MBeanTrustPermission est configurée soit avec la valeur “register”, soit avec la valeur “\*” qui signifient toutes deux la même chose dans la version courante de la version JMX.

Ainsi, utilisé en complément de la permission MBeanPermission, la permission MBeanTrustPermission permet une gestion de la sécurité plus fine sur l’enregistrement des MBeans : la permission MBeanPermission vérifie qui enregistre alors que la permission MBeanTrustPermission vérifie ce qui est enregistré.

Exemples de fichier de policy :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | grant {  permission javax.management.MBeanPermission "\*", "\*";  };  grant codeBase "file:${user.dir}${/}appl1.jar" {  permission javax.management.MBeanPermission "", "getClassLoaderRepository";  };  grant codeBase "file:${user.dir}${/}appl2.jar" {  permission javax.management.MBeanPermission "[d1:\*]", "isInstanceOf, getObjectInstance";  };  grant codeBase "file:${user.dir}${/}appl3.jar" {  permission javax.management.MBeanServerPermission "findMBeanServer";  permission javax.management.MBeanPermission "JMImplementation:\*", "queryNames";  };  grant {  permission javax.management.MBeanTrustPermission "register";  };  grant signedBy "Gorey" {  permission javax.management.MBeanTrustPermission "register";  }; |

Les connecteurs

La spécification JMX défnit la notion de connecteurs : un connecteur est attaché à l’API JMX d’un MBean Server et le rend accessible de clients Java distants, le connecteur client possèdant une interface similaire à celle du MBean Server.

Un connecteur est donc constitué d’une partie cliente et d’une partie serveur :

* Le connecteur serveur est attaché au MBean Server et écoute les requêtes de connexion des clients
* Le connecteur client a pour rôle de trouver le server et d’étable une connexion avec ce dernier. Un connecteur client se trouve généralement dans une JVM différente du connecteur serveur et s’exécutera même généralement sur une machine différente.

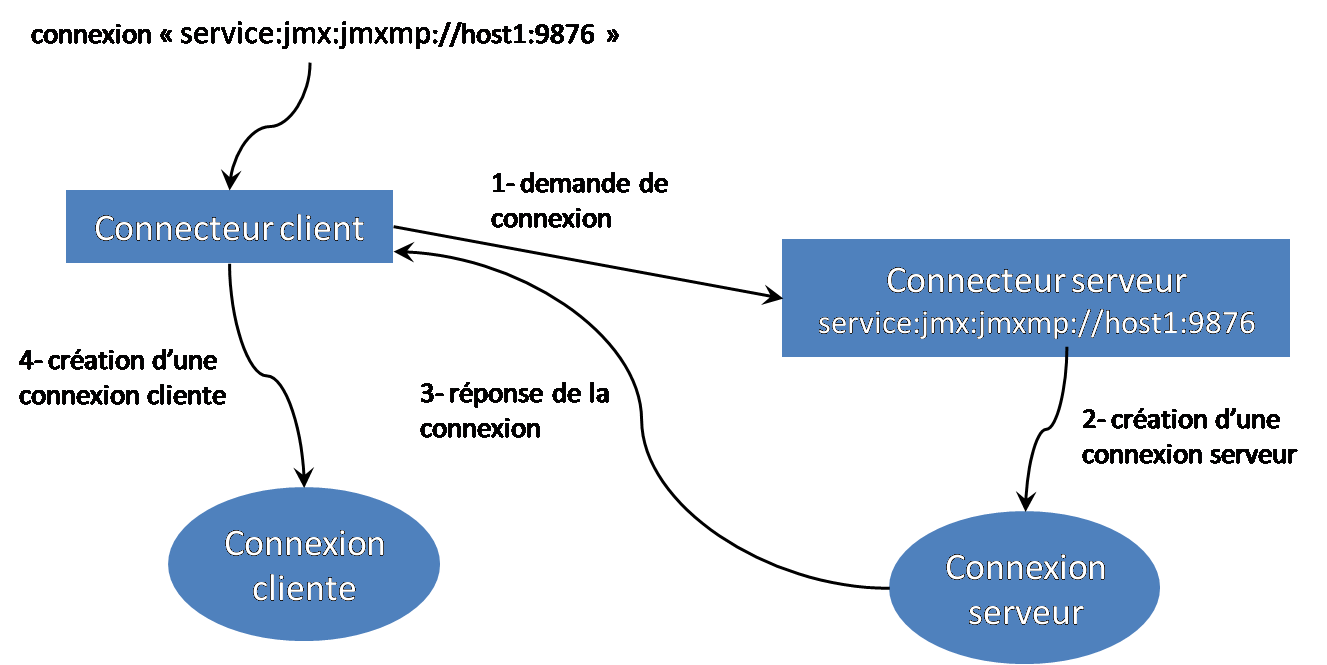
Il est à noter que contrairement à un connecteur serveur, le connecteur client ne pourra se connecter qu’à un et un seul connecteur serveur (mais une application cliente pourra posséder plusieurs connecteurs clients).

JMX permet d’avoir de multiples connecteurs qui peuvent s’appuyer sur différents protocoles de communication entre le client et le serveur. Cependant, il définit le protocole standard et obligatoire sur tous serveurs JMX s’appuyant sur RMI (*Remote Method Invocation*) et le protocole optionel s’appuyant sur des sockets TCP et appelé JMXMP (*JMX Messaging Protocol*).

Les connecteurs distinguent deux notions distinctes qui sont la **session** et la **connexion**. Un connecteur client voit une session qui peut, tout au long de son cycle de vie de la session, avoir plusieurs connexions successives à un connecteur serveur. Il est à noter que, dans certains cas, il est possible d’avoir une connexion par requête (comme cela peut être le cas avec le protocole UDP ou JMS).

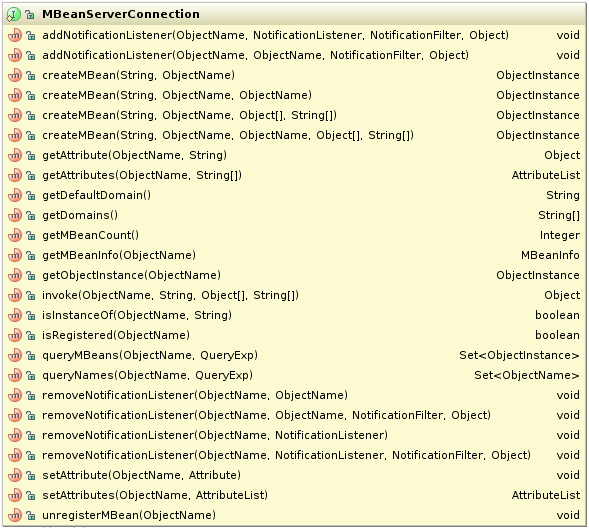
Une session possède un état sur le client mais pas nécessairement sur le connecteur serveur. Un connecteur, ne possède, quant à lui, pas forcément d’état.

Pour se connecter au connecteur serveur, le connecteur client peut utiliser une adresse du type service:jmx:jmxmp://host1:9876. Si la requête de connexion aboutie, le point d’accès client du connecteur client est retourné en réponse.



Du point de vue de la connexion cliente, le code de l’utilisateur peut obtenir un objet qui implémente l’interface MBeanServerConnection. Cette interface étant similaire à l’interface MBeanServer, il devient transparent pour le code de l’application cliente d’interagir avec un MBean Server ne se trouvant pas dans la même JVM.

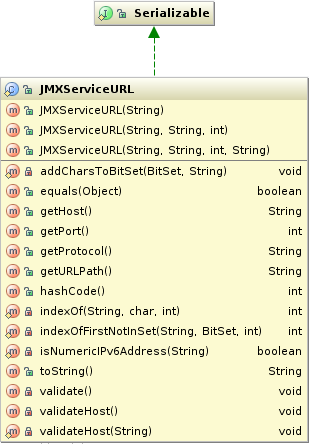
En fait, l’interface MBeanServerConnection est étendue par l’interface MBeanServer et contient donc toutes les méthodes principales sauf celles qui n’ont de sens que pour une utilisation locale.



|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Lors de l’enregistrement d’un écouteur auprès du MBeanServerConnection, le connecteur s’assure de la transmission de la notification de la partie serveur du connecteur à la partie cliente, puis à l’écouteur. Les détails de la mise en œuvre de la transmission de la notification sont dépendants du protocole. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Lors de l’utilisation d’un filtre de notifications, le filtrage des notifications reçues peut être effectué soit par la partie cliente du connecteur, soit par le MBean Server suite à la transmission du filtre. Il est, bien sûr, à noter que le filtrage des notifications coté serveur évite la transmission des notifications sur le réseau. Cependant, il est conseillé de rendre les filtres indépendants de leur lieu d’exécution (pour rappel, l’interface NotificationFilter étend l’interface Serializable). |

Toutes les implémentations des connecteurs coté serveur doivent posséder un buffer de notifications qui correspond à la liste de toutes notifications émises par les MBeans à destination du MBean Server. Il est à noter que les connecteurs doivent supporter les requêtes concurrentes pour des raisons évidentes. Concernant l’adresse du serveur connecteur, il est conseillé, pour la générer, de passer par la classe JMXServiceURL.



Afin de créer la partie cliente d’un connecteur, il existe deux possibilités :

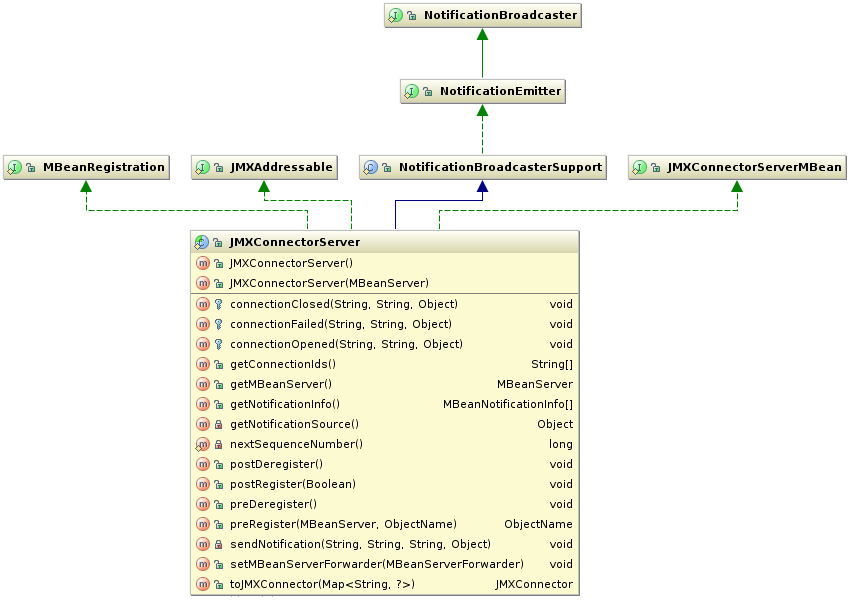
* si l’adresse du serveur est connue, la méthode connect(JMXServiceURL) de la classe statique JMXConnectorFactory peut être utilisée,
* sinon, il est possible d’interroger le serveur pour obtenir un stub de JMXConnector.

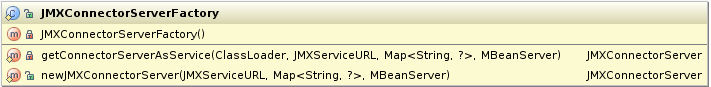
Du point de vue de la connexion serveur, la connexion serveur est représentée par une sous-classe de JMXConnectorServer qui peut être obtenue :

* soit à l’aide de la classe statique JMXConnectorServerFactory en lui indiquant le paramètre de type JMXServiceURL qui lui permet de connaitre les classes à instancier,
* soit en instanciant directement la sous-classe de JMXConnectorServer.

Le connecteur serveur doit être attaché à un MBean Server (ce qui ne signifie pas qu’il doit s’enregistré en son sein) et doit être actif pour être utilisable. Il peut être attaché auprès du MBean Server de deux manières :

* Le MBean Server auquel est attaché le connecteur serveur lui est spécifié lors de sa construction
* Le connecteur serveur est enregistré auprès du MBean Server comme s’il s’agissait d’un MBean





Exemple de création et d’attachement d’un connecteur serveur auprès d’un MBean Server

|  |  |
| --- | --- |
| 1  2  3  4 | MBeanServer mbs = MBeanServerFactory.createMBeanServer();  JMXServiceURL addr = new JMXServiceURL("jmxmp", null, 0);  JMXConnectorServer cs = JMXConnectorServerFactory.newJMXConnectorServer(addr, null, mbs);  cs.start(); |

Exemple de création et d’enregistrement d’un connecteur serveur dans un MBean Server

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | MBeanServer mbs = MBeanServerFactory.createMBeanServer();  JMXServiceURL addr = new JMXServiceURL("jmxmp", null, 0);  JMXConnectorServer cs = JMXConnectorServerFactory.newJMXConnectorServer(addr, null, null);  ObjectName csName = new ObjectName(":type=cserver,name=mycserver");  mbs.registerMBean(cs, csName);  cs.start(); |

JMX définit également comment un agent peut publier son serveur de connexion dans une infrastructure de recherche et de découverte ; ce qui permet à l’API cliente distante de savoir où trouver un serveur et ainsi de pouvoir si connecter. Cela couvre :

* SLP (*Service Location Protocol*) en décrivant comme un agent enregistre le service URL d’un connecteur serveur avec SLP
* Jini en décrivant comment un agent enregistre un stub du connecteur serveur dans le service de recherche Jini
* JNDI (*Java Naming and Directory Interface*) en décrivant un agent enregistre le service URL d’un connecteur dans un annuaire LDAP

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Un connecteur client peut spécifier un class loader par défaut lorsqu’il effectue sa connexion au serveur. Ce class loader est alors utilisé lors de la déserialisation des objets reçus du serveur que ce soit les valeurs retournées, les exceptions ou les notifications. Le class loader par défaut est positionné par la valeur de l’attribut jmx.remote.default.class.loader définie par l’environnement JMXConnector, sinon, il s’agit du class loader du contexte. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Le class loader utilisé par le connecteur serveur pour déserialiser les paramètres reçus par le connecteur client est dépendant de l’opération : en effet, parfois, il peut s’agir de celui du MBean cible (si les types de paramètres ne sont pas définis par l’API JMX), parfois, de celui configuré pendant la création du connecteur server (s’il est prévu de l’utiliser dans une application de gestion particulière qui peut définir ses propres sous classes de types de paramètres de MBean ou *NotificationFilter*). Le connecteur serveur possède un class loader par défaut qui est déterminé lors de son démarrage comme suit :   * Si l’attribut *jmx.remote.default.class.loader* de l’environnement existe, sa valeur détermine le class loader à utiliser * Si l’attribut *jmx.remote.default.class.loader.name* de l’environnement existe, sa valeur détermine l’ObjectName du MBean à utiliser comme class loader par défaut. Cette fonctionnalité permet à un connecteur serveur d’être créé avec un class loader qui est un m-let dans le même MBean Server * Sinon, c’est le class loader du thread contextuel qui est utilisé   Pour plus d’informations sur l’utilisation de class loader étendu, il est recommandé de se référer au chapitre 13.11 Class Loading de la spécification JMX. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Afin d’éviter les difficultés avec le fonctionnement des class loader Java, il est conseillé de n’utiliser que les types standards Java ou fournis par l’API JMX ou les Open MBeans qui permettent des structures plus complexes. De même, pour les notifications, il est préconisé de n’utiliser que les notifcations suivantes :   * *NotificationFilterSupport* * *MBeanServerNotificationFilter* * *AttributeChangeNotificationFilter* |

Le connecteur serveur est, en outre, capable d’authentifier le client distant. Par exemple, pour le connecteur RMI, cela est fait fournissant une implémentation de l’interface JMXAuthenticator lors de la création du connecteur serveur. Dans le cas du connecteur JMXMP, cela est fait via SASL. Dans ces deux cas, le résultat de l’authentification renvoie un Subject JAAS qui représente l’identité authentifiée. Les requêtes alors reçues du client sont exécutées en utilisant cette identité. Avec JAAS, il est possible de définir les permissions liées à une identité. En particulié, il est possible de restreindre l’accès aux opérations du MBean Server en utilisant la classe MBeanPermission. Bien sûr, pour que cela fonctionne, il est nécessaire d’avoir un SecurityManager. Dans le cas où le connecteur serveur ne supporte pas l’authentification ou qu’il n’est pas configuré, les requêtes clientes sont exécutées en utilisant la même identité que celle qui a été créée avec le connecteur serveur. Comme alternative à JAAS, il est possible de controler l’accès aux opérations du MBean Server en utilisant un MBeanServerForwarder. Cet objet est une implémentation de l’interface MBeanServer et transmet les méthodes à un autre objet MBeanServer mais en ajoutant des opérations avant et après la transmission. Cela permet, en l’occurrence d’y faire des controles d’accès. Pour ajouter un MBeanServerForwarder, la méthode setMBeanServerForwarderpeut être utilisée. De plus, pour chaque connexion à un connecteur serveur, au moins un Subject authentifié est néccessaire. Cela signifie que si un client effectue des opérations qui nécessitent plusieurs identités, des connexions différentes doivent être établies. Cependant, il est possible aux connecteurs (tels que les connecteurs RMI et JMXMP) d’utiliser la notion de délégation de Subject. Pour ce faire, avec chaque requête, le client spécifie un Subject. La requête est alors exécutée en utilisant une identité pour chaque requête si cela est possible. La permission a utilisé doit alors être de type SubjectDelegationPermission. Dans ce cas, pour chaque Subject délégué, le client obtient un MBeanServerConnection du JMXConnector pour un Subject authentifié. Les requêtes utilisées par le MBeanServerConnectionsont émises avec un Subject délégué. Il est, de plus, à noter que ces objets de type MBeanServerConnection peuvent être obtenus du même JMXConnector et peuvent être utilisés simultanément. Ainsi, pour résumer, il est possible de configurer les permissions d’un connecteur serveur de deux manières :

* toutes les permissions pour les opérations de chaque client distant doivent être configurées,
* un SubjectDelegationPermission pour chaque Principal utilisé par les clients distants doit être configuré.

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Le connecteur RMI est le seul connecteur qui doit être présent dans les implémentations de JMX. Il utilise l’infrastructure RMI (Remote Method Invocation) pour la communication entre le client et le serveur. RMI définit deux couches de transport standard :   * *Java Remote Method Protocol* (JRMP) qui est le protocole de transport par défaut * *Internet Inter-ORB Protocol* (IIOP) qui est le protocole défini par CORBA et qui permet l’interopérabilité avec un autre langage de programmation   Pour chaque serveur de connexion RMI, un objet distant implémentant l’interface distante RMIServer existe. Un client voulant communiquer avec le connecteur serveur doit obtenir une référence distante (ou stub) connecté à cet objet distant. Via RMI, chaque méthode appelée sur un stub sera transféré à l’objet distant. Ainsi, le client possédant un stub sur l’objet *RMIServer* pourra appeler obtenir le même résultat que s’il avait appelé l’objet du serveur. De plus, il existe un objet distant pour chaque connexion cliente qui permet de créer un nouvel objet distant qui implémente l’interface distante *RMIConnection*. Enfin, il est à noter que le code de l’utilisateur n’interagit généralement pas directement avec les objets du *RMIServer* ou du *RMIConnection* : en général, le stub *RMIServer* est obtenu au travers d’un objet *RMIConnector*qui implémente l’interface *JMXConnector*. L’obtention de cet objet *RMIConnector* peut être fait de trois manières :   * En fournissant un *JMXServiceURL* au *JMXConnectorFactory* qui spécifie le protocol RMI ou IIOP. * En obtenant le stub *JMXConnector* de ” quelque part ” (annuaire LDAP, Service de lookup JNDI, …). * En obtenant un stub *RMIServer* de ” quelque part ” et qui peut être utilisé comme paramètre du constructeur de *RMIConnector*.   Les notions de sécurité et de versionnement sont également décrites dans le spécification JMX (et plus particulièrement la JSR160). Cependant, ce document n’abordera pas ces points. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | La spécification JMX (et plus précisément la JSR160) décrit comment implémenter un connecteur générique en précisant comment configurer par plugin son protocole de transport ainsi que la manière dont il encapsule les objets transitant du client au serveur et plus particulièrement sa gestion du classloader du MBean cible. Cependant, ce document n’abordera pas ces points. Aussi, pour plus d’informations, il est conseillé de se référer à la JSR160 ou aux chapitres 15 - Generic Connector et 16 - Defining a new transport des spécifications agrégées JMX. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | Le connecteur JMXMP est une configuration particulière du connecteur générique où le protocole de transport s’appuie sur TCP et l’encapsulation des objets est la sérialisation native Java. |

|  |  |
| --- | --- |
| <https://3.bp.blogspot.com/_XLL8sJPQ97g/TNCS7qzv19I/AAAAAAAAAL4/6qkEYy4pX6o/s1600/remarque.png> | La spécification JMX ne spécifie pas comment trouver l’adresse d’un serveur de connecteurs attaché à un agent connu ou comment découvrir les agents présents dans le système ainsi que ce qu’ils proposent. Cependant, la JSR160 propose trois infrastructures permettant de répondre à ces besoins :   * SLP (*Service Location Protocol*) * JINI * JNDI (*Java Naming and Discovery Interface*) utilisé au travers un annuaire LDAP   Bien sûr, les APIs pour utiliser ces différentes technologies diffèrent cependant, les principes généraux restent identiques :   * L’agent créé un ou plusieurs serveurs de connecteurs * Pour chacun des connecteurs à exposer, il expose soit le *JMXServiceURL* (dans le cas de SLP ou JNDI), soit le stub *JMXConnector* (dans le cas de JINI ou JNDI) qu’il enregistre dans le service de recherche (en spécifiant au besoin d’autres méta-données) * Le client interroge le service de recherche pour obtenir un ou plusieurs *JMXServiceURL* ou stub *JMXConnector* * Le client utilise soit le factory *JMXConnectorFactory* pour obtenir un *JMXConnector* connecté au serveur s’il possède un *JMXServiceURL*, soit se connecte directement au serveur en utilisant le stub *JMXConnector*   Ce document ne fournira pas plus d’informations sur ces points. Aussi, pour plus d’informations, il est conseillé de se référer à la JSR160 ou au chapitres 17 -*Bindings to lookup services* des spécifications agrégées JMX. |

Le mot de la fin

Voilà, ce dernier article met fin à cette série. Si vous avez réussi à tenir jusque là et à tout lire, félicitation! Maintenant, vous devriez avoir (enfin, j’espère) une vision plus précise sur ce que contiennent les spécifications JMX. Bien sûr, cette série n’était qu’un condensée (qui vaut ce qu’il vaut) permettant de comprendre les concepts sans pour autant lire les 290 pages de spécification et ne se veut pas être suffisante pour tout maîtriser.