| | [**Overview**](http://docs.google.com/overview-summary.html) | [**Package**](http://docs.google.com/package-summary.html) | **Class** | [**Use**](http://docs.google.com/class-use/MXBean.html) | [**Tree**](http://docs.google.com/package-tree.html) | [**Deprecated**](http://docs.google.com/deprecated-list.html) | [**Index**](http://docs.google.com/index-files/index-1.html) | [**Help**](http://docs.google.com/help-doc.html) | | --- | --- | --- | --- | --- | --- | --- | --- | | | ***Java™ Platform***  ***Standard Ed. 6*** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**PREV CLASS**](http://docs.google.com/javax/management/MBeanTrustPermission.html)   [**NEXT CLASS**](http://docs.google.com/javax/management/NotCompliantMBeanException.html) | [**FRAMES**](http://docs.google.com/index.html?javax/management/MXBean.html)    [**NO FRAMES**](http://docs.google.com/MXBean.html)     [**All Classes**](http://docs.google.com/allclasses-noframe.html) |
| SUMMARY: REQUIRED | [OPTIONAL](#17dp8vu) | DETAIL: [ELEMENT](#1ksv4uv) |

## **javax.management**

Annotation Type MXBean

[@Documented](http://docs.google.com/java/lang/annotation/Documented.html)  
[@Retention](http://docs.google.com/java/lang/annotation/Retention.html)([value](http://docs.google.com/java/lang/annotation/Retention.html#value())=[RUNTIME](http://docs.google.com/java/lang/annotation/RetentionPolicy.html#RUNTIME))  
[@Target](http://docs.google.com/java/lang/annotation/Target.html)([value](http://docs.google.com/java/lang/annotation/Target.html#value())=[TYPE](http://docs.google.com/java/lang/annotation/ElementType.html#TYPE))  
public @interface **MXBean**

Annotation to mark an interface explicitly as being an MXBean interface, or as not being an MXBean interface. By default, an interface is an MXBean interface if its name ends with MXBean, as in SomethingMXBean. The following interfaces are MXBean interfaces:

public interface WhatsitMXBean {}  
  
 @MXBean  
 public interface Whatsit1Interface {}  
  
 @MXBean(true)  
 public interface Whatsit2Interface {}

The following interfaces are not MXBean interfaces:

public interface Whatsit3Interface{}  
  
 @MXBean(false)  
 public interface MisleadingMXBean {}

### MXBean specification

The MXBean concept provides a simple way to code an MBean that only references a predefined set of types, the ones defined by [javax.management.openmbean](http://docs.google.com/javax/management/openmbean/package-summary.html). In this way, you can be sure that your MBean will be usable by any client, including remote clients, without any requirement that the client have access to *model-specific classes* representing the types of your MBeans.

The concepts are easier to understand by comparison with the Standard MBean concept. Here is how a managed object might be represented as a Standard MBean, and as an MXBean:

| Standard MBean | MXBean |
| --- | --- |
| public interface MemoryPool**MBean** {  String getName();  MemoryUsage getUsage();  // ... } | public interface MemoryPool**MXBean** {  String getName();  MemoryUsage getUsage();  // ... } |

As you can see, the definitions are very similar. The only difference is that the convention for naming the interface is to use *Something*MXBean for MXBeans, rather than *Something*MBean for Standard MBeans.

In this managed object, there is an attribute called Usage of type [MemoryUsage](http://docs.google.com/java/lang/management/MemoryUsage.html). The point of an attribute like this is that it gives a coherent snapshot of a set of data items. For example, it might include the current amount of used memory in the memory pool, and the current maximum of the memory pool. If these were separate items, obtained with separate [getAttribute](http://docs.google.com/javax/management/MBeanServer.html#getAttribute(javax.management.ObjectName,%20java.lang.String)) calls, then we could get values seen at different times that were not consistent. We might get a used value that was greater than the max value.

So, we might define MemoryUsage like this:

| Standard MBean | MXBean |
| --- | --- |
| public class MemoryUsage **implements Serializable** {  // standard JavaBean conventions with getters   public MemoryUsage(long init, long used,  long committed, long max) {...}  long getInit() {...}  long getUsed() {...}  long getCommitted() {...}  long getMax() {...} } | public class MemoryUsage {  // standard JavaBean conventions with getters  **@ConstructorProperties({"init", "used", "committed", "max"})**  public MemoryUsage(long init, long used,  long committed, long max) {...}  long getInit() {...}  long getUsed() {...}  long getCommitted() {...}  long getMax() {...} } |

The definitions are the same in the two cases, except that with the MXBean, MemoryUsage no longer needs to be marked Serializable (though it can be). On the other hand, we have added a @ConstructorProperties annotation to link the constructor parameters to the corresponding getters. We will see more about this below.

MemoryUsage is a *model-specific class*. With Standard MBeans, a client of the MBean Server cannot access the Usage attribute if it does not know the class MemoryUsage. Suppose the client is a generic console based on JMX technology. Then the console would have to be configured with the model-specific classes of every application it might connect to. The problem is even worse for clients that are not written in the Java language. Then there may not be any way to tell the client what a MemoryUsage looks like.

This is where MXBeans differ from Standard MBeans. Although we define the management interface in almost exactly the same way, the MXBean framework *converts* model-specific classes into standard classes from the Java platform. Using arrays and the [CompositeData](http://docs.google.com/javax/management/openmbean/CompositeData.html) and [TabularData](http://docs.google.com/javax/management/openmbean/TabularData.html) classes from the standard [javax.management.openmbean](http://docs.google.com/javax/management/openmbean/package-summary.html) package, it is possible to build data structures of arbitrary complexity using only standard classes.

This becomes clearer if we compare what the clients of the two models might look like:

| Standard MBean | MXBean |
| --- | --- |
| String name = (String)  mbeanServer.[getAttribute](http://docs.google.com/javax/management/MBeanServer.html#getAttribute(javax.management.ObjectName,%20java.lang.String))(objectName, "Name"); **MemoryUsage** usage = (**MemoryUsage**)  mbeanServer.getAttribute(objectName, "Usage"); **long used = usage.getUsed();** | String name = (String)  mbeanServer.[getAttribute](http://docs.google.com/javax/management/MBeanServer.html#getAttribute(javax.management.ObjectName,%20java.lang.String))(objectName, "Name"); [**CompositeData**](http://docs.google.com/javax/management/openmbean/CompositeData.html) usage = (**CompositeData**)  mbeanServer.getAttribute(objectName, "Usage"); **long used = (Long) usage.**[**get**](http://docs.google.com/javax/management/openmbean/CompositeData.html#get(java.lang.String))**("used");** |

For attributes with simple types like String, the code is the same. But for attributes with complex types, the Standard MBean code requires the client to know the model-specific class MemoryUsage, while the MXBean code requires no non-standard classes.

The client code shown here is slightly more complicated for the MXBean client. But, if the client does in fact know the model, here the interface MemoryPoolMXBean and the class MemoryUsage, then it can construct a *proxy*. This is the recommended way to interact with managed objects when you know the model beforehand, regardless of whether you are using Standard MBeans or MXBeans:

| Standard MBean | MXBean |
| --- | --- |
| MemoryPool**MBean** proxy =  JMX.[**newMBeanProxy**](http://docs.google.com/javax/management/JMX.html#newMBeanProxy(javax.management.MBeanServerConnection,%20javax.management.ObjectName,%20java.lang.Class))(  mbeanServer,  objectName,  MemoryPool**MBean**.class); String name = proxy.getName(); MemoryUsage usage = proxy.getUsage(); long used = usage.getUsed(); | MemoryPool**MXBean** proxy =  JMX.[**newMXBeanProxy**](http://docs.google.com/javax/management/JMX.html#newMXBeanProxy(javax.management.MBeanServerConnection,%20javax.management.ObjectName,%20java.lang.Class))(  mbeanServer,  objectName,  MemoryPool**MXBean**.class); String name = proxy.getName(); MemoryUsage usage = proxy.getUsage(); long used = usage.getUsed(); |

Implementing the MemoryPool object works similarly for both Standard MBeans and MXBeans.

| Standard MBean | MXBean |
| --- | --- |
| public class MemoryPool  implements MemoryPool**MBean** {  public String getName() {...}  public MemoryUsage getUsage() {...}  // ... } | public class MemoryPool  implements MemoryPool**MXBean** {  public String getName() {...}  public MemoryUsage getUsage() {...}  // ... } |

Registering the MBean in the MBean Server works in the same way in both cases:

| Standard MBean | MXBean |
| --- | --- |
| {  MemoryPool**MBean** pool = new MemoryPool();  mbeanServer.[registerMBean](http://docs.google.com/javax/management/MBeanServer.html#registerMBean(java.lang.Object,%20javax.management.ObjectName))(pool, objectName); } | {  MemoryPool**MXBean** pool = new MemoryPool();  mbeanServer.[registerMBean](http://docs.google.com/javax/management/MBeanServer.html#registerMBean(java.lang.Object,%20javax.management.ObjectName))(pool, objectName); } |

## Definition of an MXBean

An MXBean is a kind of MBean. An MXBean object can be registered directly in the MBean Server, or it can be used as an argument to [StandardMBean](http://docs.google.com/javax/management/StandardMBean.html) and the resultant MBean registered in the MBean Server.

When an object is registered in the MBean Server using the registerMBean or createMBean methods of the [MBeanServer](http://docs.google.com/javax/management/MBeanServer.html) interface, the object's class is examined to determine what type of MBean it is:

* If the class implements the interface [DynamicMBean](http://docs.google.com/javax/management/DynamicMBean.html) then the MBean is a Dynamic MBean. Note that the class StandardMBean implements this interface, so this case applies to a Standard MBean or MXBean created using the class StandardMBean.
* Otherwise, if the class matches the Standard MBean naming conventions, then the MBean is a Standard MBean.
* Otherwise, it may be an MXBean. The set of interfaces implemented by the object is examined for interfaces that:
  + have a class name *S*MXBean where *S* is any non-empty string, and do not have an annotation @MXBean(false); and/or
  + have an annotation @MXBean(true) or just @MXBean.

If there is exactly one such interface, or if there is one such interface that is a subinterface of all the others, then the object is an MXBean. The interface in question is the *MXBean interface*. In the example above, the MXBean interface is MemoryPoolMXBean.

* If none of these conditions is met, the MBean is invalid and the attempt to register it will generate [NotCompliantMBeanException](http://docs.google.com/javax/management/NotCompliantMBeanException.html).

Every Java type that appears as the parameter or return type of a method in an MXBean interface must be *convertible* using the rules below. Additionally, parameters must be *reconstructible* as defined below.

An attempt to construct an MXBean that does not conform to the above rules will produce an exception.

## Naming conventions

The same naming conventions are applied to the methods in an MXBean as in a Standard MBean:

1. A method *T* get*N*(), where *T* is a Java type (not void) and *N* is a non-empty string, specifies that there is a readable attribute called *N*. The Java type and Open type of the attribute are determined by the mapping rules below. The method final Class getClass() inherited from Object is ignored when looking for getters.
2. A method boolean is*N*() specifies that there is a readable attribute called *N* with Java type boolean and Open type SimpleType.Boolean.
3. A method void set*N*(*T* x) specifies that there is a writeable attribute called *N*. The Java type and Open type of the attribute are determined by the mapping rules below. (Of course, the name x of the parameter is irrelevant.)
4. Every other method specifies that there is an operation with the same name as the method. The Java type and Open type of the return value and of each parameter are determined by the mapping rules below.

The rules for get*N* and is*N* collectively define the notion of a *getter*. The rule for set*N* defines the notion of a *setter*.

It is an error for there to be two getters with the same name, or two setters with the same name. If there is a getter and a setter for the same name, then the type *T* in both must be the same. In this case the attribute is read/write. If there is only a getter or only a setter, the attribute is read-only or write-only respectively.

## Type mapping rules

An MXBean is a kind of Open MBean, as defined by the [javax.management.openmbean](http://docs.google.com/javax/management/openmbean/package-summary.html) package. This means that the types of attributes, operation parameters, and operation return values must all be describable using *Open Types*, that is the four standard subclasses of [OpenType](http://docs.google.com/javax/management/openmbean/OpenType.html). MXBeans achieve this by mapping Java types into Open Types.

For every Java type *J*, the MXBean mapping is described by the following information:

* The corresponding Open Type, *opentype(J)*. This is an instance of a subclass of [OpenType](http://docs.google.com/javax/management/openmbean/OpenType.html).
* The *mapped* Java type, *opendata(J)*, which is always the same for any given *opentype(J)*. This is a Java class.
* How a value is converted from type *J* to type *opendata(J)*.
* How a value is converted from type *opendata(J)* to type *J*, if it can be.

For example, for the Java type List<String>:

* The Open Type, *opentype(*List<String>*)*, is [ArrayType](http://docs.google.com/javax/management/openmbean/ArrayType.html)(1, [SimpleType.STRING](http://docs.google.com/javax/management/openmbean/SimpleType.html#STRING)), representing a 1-dimensional array of Strings.
* The mapped Java type, *opendata(*List<String>*)*, is String[].
* A List<String> can be converted to a String[] using [List.toArray(new String[0])](http://docs.google.com/java/util/List.html#toArray(T%5B%5D)).
* A String[] can be converted to a List<String> using [Arrays.asList](http://docs.google.com/java/util/Arrays.html#asList(T...)).

If no mapping rules exist to derive *opentype(J)* from *J*, then *J* cannot be the type of a method parameter or return value in an MXBean interface.

If there is a way to convert *opendata(J)* back to *J* then we say that *J* is *reconstructible*. All method parameters in an MXBean interface must be reconstructible, because when the MXBean framework is invoking a method it will need to convert those parameters from *opendata(J)* to *J*. In a proxy generated by [JMX.newMXBeanProxy](http://docs.google.com/javax/management/JMX.html#newMXBeanProxy(javax.management.MBeanServerConnection,%20javax.management.ObjectName,%20java.lang.Class)), it is the return values of the methods in the MXBean interface that must be reconstructible.

Null values are allowed for all Java types and Open Types, except primitive Java types where they are not possible. When converting from type *J* to type *opendata(J)* or from type *opendata(J)* to type *J*, a null value is mapped to a null value.

The following table summarizes the type mapping rules.

| Java type *J* | *opentype(J)* | *opendata(J)* |
| --- | --- | --- |
| int, boolean, etc  (the 8 primitive Java types) | SimpleType.INTEGER,  SimpleType.BOOLEAN, etc | Integer, Boolean, etc  (the corresponding boxed types) |
| Integer, ObjectName, etc  (the types covered by [SimpleType](http://docs.google.com/javax/management/openmbean/SimpleType.html)) | the corresponding SimpleType | *J*, the same type |
| int[] etc  (a one-dimensional array with  primitive element type) | ArrayType.getPrimitiveArrayType(int[].class) etc | *J*, the same type |
| *E*[]  (an array with non-primitive element type *E*; this includes int[][], where *E* is int[]) | ArrayType.getArrayType(*opentype(E)*) | *opendata(E)*[] |
| List<*E*>  Set<*E*>  SortedSet<*E*> (see below) | same as for *E*[] | same as for *E*[] |
| An enumeration *E*  (declared in Java as enum *E* {...}) | SimpleType.STRING | String |
| Map<*K*,*V*>  SortedMap<*K*,*V*> | [TabularType](http://docs.google.com/javax/management/openmbean/TabularType.html)  (see below) | [TabularData](http://docs.google.com/javax/management/openmbean/TabularData.html)  (see below) |
| An MXBean interface | SimpleType.OBJECTNAME  (see below) | [ObjectName](http://docs.google.com/javax/management/ObjectName.html)  (see below) |
| Any other type | [CompositeType](http://docs.google.com/javax/management/openmbean/CompositeType.html), if possible  (see below) | [CompositeData](http://docs.google.com/javax/management/openmbean/CompositeData.html) |

The following sections give further details of these rules.

### Mappings for primitive types

The 8 primitive Java types (boolean, byte, short, int, long, float, double, char) are mapped to the corresponding boxed types from java.lang, namely Boolean, Byte, etc. The Open Type is the corresponding SimpleType. Thus, *opentype(*long*)* is SimpleType.LONG, and *opendata(*long*)* is java.lang.Long.

An array of primitive type such as long[] can be represented directly as an Open Type. Thus, *openType(*long[]*)* is ArrayType.getPrimitiveArrayType(long[].class), and *opendata(*long[]*)* is long[].

In practice, the difference between a plain int and Integer, etc, does not show up because operations in the JMX API are always on Java objects, not primitives. However, the difference *does* show up with arrays.

### Mappings for collections (List<***E***> etc)

A List<*E*> or Set<*E*>, such as List<String> or Set<ObjectName>, is mapped in the same way as an array of the same element type, such as String[] or ObjectName[].

A SortedSet<*E*> is also mapped in the same way as an *E*[], but it is only convertible if *E* is a class or interface that implements [Comparable](http://docs.google.com/java/lang/Comparable.html). Thus, a SortedSet<String> or SortedSet<Integer> is convertible, but a SortedSet<int[]> or SortedSet<List<String>> is not. The conversion of a SortedSet instance will fail with an IllegalArgumentException if it has a non-null [comparator()](http://docs.google.com/java/util/SortedSet.html#comparator()).

A List<*E*> is reconstructed as a java.util.ArrayList<*E*>; a Set<*E*> as a java.util.HashSet<*E*>; a SortedSet<*E*> as a java.util.TreeSet<*E*>.

### Mappings for maps (Map<***K***,***V***> etc)

A Map<*K*,*V*> or SortedMap<*K*,*V*>, for example Map<String,ObjectName>, has Open Type [TabularType](http://docs.google.com/javax/management/openmbean/TabularType.html) and is mapped to a [TabularData](http://docs.google.com/javax/management/openmbean/TabularData.html). The TabularType has two items called key and value. The Open Type of key is *opentype(K)*, and the Open Type of value is *opentype(V)*. The index of the TabularType is the single item key.

For example, the TabularType for a Map<String,ObjectName> might be constructed with code like this:

String typeName =  
 "java.util.Map<java.lang.String, javax.management.ObjectName>";  
String[] keyValue =  
 new String[] {"key", "value"};  
OpenType[] openTypes =  
 new OpenType[] {SimpleType.STRING, SimpleType.OBJECTNAME};  
CompositeType rowType =  
 new CompositeType(typeName, typeName, keyValue, keyValue, openTypes);  
TabularType tabularType =  
 new TabularType(typeName, typeName, rowType, new String[] {"key"});

The typeName here is determined by the  [type name rules](#_2s8eyo1) detailed below.

A SortedMap<*K*,*V*> is mapped in the same way, but it is only convertible if *K* is a class or interface that implements [Comparable](http://docs.google.com/java/lang/Comparable.html). Thus, a SortedMap<String,int[]> is convertible, but a SortedMap<int[],String> is not. The conversion of a SortedMap instance will fail with an IllegalArgumentException if it has a non-null [comparator()](http://docs.google.com/java/util/SortedMap.html#comparator()).

A Map<*K*,*V*> is reconstructed as a java.util.HashMap<*K*,*V*>; a SortedMap<*K*,*V*> as a java.util.TreeMap<*K*,*V*>.

TabularData is an interface. The concrete class that is used to represent a Map<*K*,*V*> as Open Data is [TabularDataSupport](http://docs.google.com/javax/management/openmbean/TabularDataSupport.html), or another class implementing TabularData that serializes as TabularDataSupport.

### Mappings for MXBean interfaces

An MXBean interface, or a type referenced within an MXBean interface, can reference another MXBean interface, *J*. Then *opentype(J)* is SimpleType.OBJECTNAME and *opendata(J)* is ObjectName.

For example, suppose you have two MXBean interfaces like this:

public interface ProductMXBean {  
 public ModuleMXBean[] getModules();  
}  
  
public interface ModuleMXBean {  
 public ProductMXBean getProduct();  
}

The object implementing the ModuleMXBean interface returns from its getProduct method an object implementing the ProductMXBean interface. The ModuleMXBean object and the returned ProductMXBean objects must both be registered as MXBeans in the same MBean Server.

The method ModuleMXBean.getProduct() defines an attribute called Product. The Open Type for this attribute is SimpleType.OBJECTNAME, and the corresponding ObjectName value will be the name under which the referenced ProductMXBean is registered in the MBean Server.

If you make an MXBean proxy for a ModuleMXBean and call its getProduct() method, the proxy will map the ObjectName back into a ProductMXBean by making another MXBean proxy. More formally, when a proxy made with [JMX.newMXBeanProxy(mbeanServerConnection, objectNameX, interfaceX)](http://docs.google.com/javax/management/JMX.html#newMXBeanProxy(javax.management.MBeanServerConnection,%20javax.management.ObjectName,%20java.lang.Class)) needs to map objectNameY back into interfaceY, another MXBean interface, it does so with JMX.newMXBeanProxy(mbeanServerConnection, objectNameY, interfaceY). The implementation may return a proxy that was previously created by a call to JMX.newMXBeanProxy with the same parameters, or it may create a new proxy.

The reverse mapping is illustrated by the following change to the ModuleMXBean interface:

public interface ModuleMXBean {  
 public ProductMXBean getProduct();  
 public void setProduct(ProductMXBean c);  
}

The presence of the setProduct method now means that the Product attribute is read/write. As before, the value of this attribute is an ObjectName. When the attribute is set, the ObjectName must be converted into the ProductMXBean object that the setProduct method expects. This object will be an MXBean proxy for the given ObjectName in the same MBean Server.

If you make an MXBean proxy for a ModuleMXBean and call its setProduct method, the proxy will map its ProductMXBean argument back into an ObjectName. This will only work if the argument is in fact another proxy, for a ProductMXBean in the same MBeanServerConnection. The proxy can have been returned from another proxy (like ModuleMXBean.getProduct() which returns a proxy for a ProductMXBean); or it can have been created by [JMX.newMXBeanProxy](http://docs.google.com/javax/management/JMX.html#newMXBeanProxy(javax.management.MBeanServerConnection,%20javax.management.ObjectName,%20java.lang.Class)); or it can have been created using [Proxy](http://docs.google.com/java/lang/reflect/Proxy.html) with an invocation handler that is [MBeanServerInvocationHandler](http://docs.google.com/javax/management/MBeanServerInvocationHandler.html) or a subclass.

If the same MXBean were registered under two different ObjectNames, a reference to that MXBean from another MXBean would be ambiguous. Therefore, if an MXBean object is already registered in an MBean Server and an attempt is made to register it in the same MBean Server under another name, the result is an [InstanceAlreadyExistsException](http://docs.google.com/javax/management/InstanceAlreadyExistsException.html). Registering the same MBean object under more than one name is discouraged in general, notably because it does not work well for MBeans that are [NotificationBroadcaster](http://docs.google.com/javax/management/NotificationBroadcaster.html)s.

### Mappings for other types

Given a Java class or interface *J* that does not match the other rules in the table above, the MXBean framework will attempt to map it to a [CompositeType](http://docs.google.com/javax/management/openmbean/CompositeType.html) as follows. The type name of this CompositeType is determined by the  [type name rules](#_2s8eyo1) below.

The class is examined for getters using the conventions [above](#_tyjcwt). (Getters must be public instance methods.) If there are no getters, or if any getter has a type that is not convertible, then *J* is not convertible.

If there is at least one getter and every getter has a convertible type, then *opentype(J)* is a CompositeType with one item for every getter. If the getter is

*T* get*Name*()then the item in the CompositeType is called name and has type *opentype(T)*. For example, if the item isString getOwner()then the item is called owner and has Open Type SimpleType.STRING. If the getter isboolean is*Name*()then the item in the CompositeType is called name and has type SimpleType.BOOLEAN.

Notice that the first character (or code point) is converted to lower case. This follows the Java Beans convention, which for historical reasons is different from the Standard MBean convention. In a Standard MBean or MXBean interface, a method getOwner defines an attribute called Owner, while in a Java Bean or mapped CompositeType, a method getOwner defines a property or item called owner.

If two methods produce the same item name (for example, getOwner and isOwner, or getOwner and getowner) then the type is not convertible.

When the Open Type is CompositeType, the corresponding mapped Java type (*opendata(J)*) is [CompositeData](http://docs.google.com/javax/management/openmbean/CompositeData.html). The mapping from an instance of *J* to a CompositeData corresponding to the CompositeType just described is done as follows. First, if *J* implements the interface [CompositeDataView](http://docs.google.com/javax/management/openmbean/CompositeDataView.html), then that interface's [toCompositeData](http://docs.google.com/javax/management/openmbean/CompositeDataView.html#toCompositeData(javax.management.openmbean.CompositeType)) method is called to do the conversion. Otherwise, the CompositeData is constructed by calling the getter for each item and converting it to the corresponding Open Data type. Thus, a getter such as

List<String> getNames()

will have been mapped to an item with name "names" and Open Type ArrayType(1, SimpleType.STRING). The conversion to CompositeData will call getNames() and convert the resultant List<String> into a String[] for the item "names".

CompositeData is an interface. The concrete class that is used to represent a type as Open Data is [CompositeDataSupport](http://docs.google.com/javax/management/openmbean/CompositeDataSupport.html), or another class implementing CompositeData that serializes as CompositeDataSupport.

#### Reconstructing an instance of Java type ***J*** from a CompositeData

If *opendata(J)* is CompositeData for a Java type *J*, then either an instance of *J* can be reconstructed from a CompositeData, or *J* is not reconstructible. If any item in the CompositeData is not reconstructible, then *J* is not reconstructible either.

For any given *J*, the following rules are consulted to determine how to reconstruct instances of *J* from CompositeData. The first applicable rule in the list is the one that will be used.

1. If *J* has a method  
   public static *J* from(CompositeData cd)  
   then that method is called to reconstruct an instance of *J*.
2. Otherwise, if *J* has at least one public constructor with a [ConstructorProperties](http://docs.google.com/java/beans/ConstructorProperties.html) annotation, then one of those constructors (not necessarily always the same one) will be called to reconstruct an instance of *J*. Every such annotation must list as many strings as the constructor has parameters; each string must name a property corresponding to a getter of *J*; and the type of this getter must be the same as the corresponding constructor parameter. It is not an error for there to be getters that are not mentioned in the ConstructorProperties annotation (these may correspond to information that is not needed to reconstruct the object).  
   An instance of *J* is reconstructed by calling a constructor with the appropriate reconstructed items from the CompositeData. The constructor to be called will be determined at runtime based on the items actually present in the CompositeData, given that this CompositeData might come from an earlier version of *J* where not all the items were present. A constructor is *applicable* if all the properties named in its ConstructorProperties annotation are present as items in the CompositeData. If no constructor is applicable, then the attempt to reconstruct *J* fails.  
   For any possible combination of properties, it must be the case that either (a) there are no applicable constructors, or (b) there is exactly one applicable constructor, or (c) one of the applicable constructors names a proper superset of the properties named by each other applicable constructor. (In other words, there should never be ambiguity over which constructor to choose.) If this condition is not true, then *J* is not reconstructible.
3. Otherwise, if *J* has a public no-arg constructor, and for every getter in *J* with type *T* and name *N* there is a corresponding setter with the same name and type, then an instance of *J* is constructed with the no-arg constructor and the setters are called with the reconstructed items from the CompositeData to restore the values. For example, if there is a method  
   public List<String> getNames()  
   then there must also be a method  
   public void setNames(List<String> names)  
   for this rule to apply.  
   If the CompositeData came from an earlier version of *J*, some items might not be present. In this case, the corresponding setters will not be called.
4. Otherwise, if *J* is an interface that has no methods other than getters, an instance of *J* is constructed using a [Proxy](http://docs.google.com/java/lang/reflect/Proxy.html) with a [CompositeDataInvocationHandler](http://docs.google.com/javax/management/openmbean/CompositeDataInvocationHandler.html) backed by the CompositeData being converted.
5. Otherwise, *J* is not reconstructible.

Here are examples showing different ways to code a type NamedNumber that consists of an int and a String. In each case, the CompositeType looks like this:

[CompositeType](http://docs.google.com/javax/management/openmbean/CompositeType.html)(  
 "NamedNumber", // typeName  
 "NamedNumber", // description  
 new String[] {"number", "name"}, // itemNames  
 new String[] {"number", "name"}, // itemDescriptions  
 new OpenType[] {SimpleType.INTEGER,  
 SimpleType.STRING} // itemTypes  
);

1. Static from method:public class NamedNumber {  
    public int getNumber() {return number;}  
    public String getName() {return name;}  
    private NamedNumber(int number, String name) {  
    this.number = number;  
    this.name = name;  
    }  
    **public static NamedNumber from(CompositeData cd)** {  
    return new NamedNumber((Integer) cd.get("number"),  
    (String) cd.get("name"));  
    }  
    private final int number;  
    private final String name;  
   }
2. Public constructor with @ConstructorProperties annotation:public class NamedNumber {  
    public int getNumber() {return number;}  
    public String getName() {return name;}  
    **@ConstructorProperties({"number", "name"})  
    public NamedNumber(int number, String name)** {  
    this.number = number;  
    this.name = name;  
    }  
    private final int number;  
    private final String name;  
   }
3. Setter for every getter:public class NamedNumber {  
    public int getNumber() {return number;}  
    public void **setNumber**(int number) {this.number = number;}  
    public String getName() {return name;}  
    public void **setName**(String name) {this.name = name;}  
    **public NamedNumber()** {}  
    private int number;  
    private String name;  
   }
4. Interface with only getters:public interface NamedNumber {  
    public int getNumber();  
    public String getName();  
   }

It is usually better for classes that simply represent a collection of data to be *immutable*. An instance of an immutable class cannot be changed after it has been constructed. Notice that CompositeData itself is immutable. Immutability has many advantages, notably with regard to thread-safety and security. So the approach using setters should generally be avoided if possible.

### Recursive types

Recursive (self-referential) types cannot be used in MXBean interfaces. This is a consequence of the immutability of [CompositeType](http://docs.google.com/javax/management/openmbean/CompositeType.html). For example, the following type could not be the type of an attribute, because it refers to itself:

public interface **Node** {  
 public String getName();  
 public int getPriority();  
 public **Node** getNext();  
}

It is always possible to rewrite recursive types like this so they are no longer recursive. Doing so may require introducing new types. For example:

public interface **NodeList** {  
 public List<Node> getNodes();  
}  
  
public interface Node {  
 public String getName();  
 public int getPriority();  
}

### MBeanInfo contents for an MXBean

An MXBean is a type of Open MBean. However, for compatibility reasons, its [MBeanInfo](http://docs.google.com/javax/management/MBeanInfo.html) is not an [OpenMBeanInfo](http://docs.google.com/javax/management/openmbean/OpenMBeanInfo.html). In particular, when the type of an attribute, parameter, or operation return value is a primitive type such as int, or is void (for a return type), then the attribute, parameter, or operation will be represented respectively by an [MBeanAttributeInfo](http://docs.google.com/javax/management/MBeanAttributeInfo.html), [MBeanParameterInfo](http://docs.google.com/javax/management/MBeanParameterInfo.html), or [MBeanOperationInfo](http://docs.google.com/javax/management/MBeanOperationInfo.html) whose getType() or getReturnType() returns the primitive name ("int" etc). This is so even though the mapping rules above specify that the *opendata* mapping is the wrapped type (Integer etc).

The array of public constructors returned by [MBeanInfo.getConstructors()](http://docs.google.com/javax/management/MBeanInfo.html#getConstructors()) for an MXBean that is directly registered in the MBean Server will contain all of the public constructors of that MXBean. If the class of the MXBean is not public then its constructors are not considered public either. The list returned for an MXBean that is constructed using the [StandardMBean](http://docs.google.com/javax/management/StandardMBean.html) class is derived in the same way as for Standard MBeans. Regardless of how the MXBean was constructed, its constructor parameters are not subject to MXBean mapping rules and do not have a corresponding OpenType.

The array of notification types returned by [MBeanInfo.getNotifications()](http://docs.google.com/javax/management/MBeanInfo.html#getNotifications()) for an MXBean that is directly registered in the MBean Server will be empty if the MXBean does not implement the [NotificationBroadcaster](http://docs.google.com/javax/management/NotificationBroadcaster.html) interface. Otherwise, it will be the result of calling [NotificationBroadcaster.getNotificationInfo()](http://docs.google.com/javax/management/NotificationBroadcaster.html#getNotificationInfo()) at the time the MXBean was registered. Even if the result of this method changes subsequently, the result of MBeanInfo.getNotifications() will not. The list returned for an MXBean that is constructed using the [StandardMBean](http://docs.google.com/javax/management/StandardMBean.html) or [StandardEmitterMBean](http://docs.google.com/javax/management/StandardEmitterMBean.html) class is derived in the same way as for Standard MBeans.

The [Descriptor](http://docs.google.com/javax/management/Descriptor.html) for all of the MBeanAttributeInfo, MBeanParameterInfo, and MBeanOperationInfo objects contained in the MBeanInfo will have a field openType whose value is the [OpenType](http://docs.google.com/javax/management/openmbean/OpenType.html) specified by the mapping rules above. So even when getType() is "int", getDescriptor().getField("openType") will be [SimpleType.INTEGER](http://docs.google.com/javax/management/openmbean/SimpleType.html#INTEGER).

The Descriptor for each of these objects will also have a field originalType that is a string representing the Java type that appeared in the MXBean interface. The format of this string is described in the section [Type Names](#_2s8eyo1) below.

The Descriptor for the MBeanInfo will have a field mxbean whose value is the string "true".

### Type Names

Sometimes the unmapped type *T* of a method parameter or return value in an MXBean must be represented as a string. If *T* is a non-generic type, this string is the value returned by [Class.getName()](http://docs.google.com/java/lang/Class.html#getName()). Otherwise it is the value of *genericstring(T)*, defined as follows:

* If *T* is a non-generic non-array type, *genericstring(T)* is the value returned by [Class.getName()](http://docs.google.com/java/lang/Class.html#getName()), for example "int" or "java.lang.String".
* If *T* is an array *E[]*, *genericstring(T)* is *genericstring(E)* followed by "[]". For example, *genericstring(int[])* is "int[]", and *genericstring(List<String>[][])* is "java.util.List<java.lang.String>[][]".
* Otherwise, *T* is a parameterized type such as List<String> and *genericstring(T)* consists of the following: the fully-qualified name of the parameterized type as returned by Class.getName(); a left angle bracket ("<"); *genericstring(A)* where *A* is the first type parameter; if there is a second type parameter *B* then ", " (a comma and a single space) followed by *genericstring(B)*; a right angle bracket (">").

Note that if a method returns int[], this will be represented by the string "[I" returned by Class.getName(), but if a method returns List<int[]>, this will be represented by the string "java.util.List<int[]>".

### Exceptions

A problem with mapping *from* Java types *to* Open types is signaled with an [OpenDataException](http://docs.google.com/javax/management/openmbean/OpenDataException.html). This can happen when an MXBean interface is being analyzed, for example if it references a type like [java.util.Random](http://docs.google.com/java/util/Random.html) that has no getters. Or it can happen when an instance is being converted (a return value from a method in an MXBean or a parameter to a method in an MXBean proxy), for example when converting from SortedSet<String> to String[] if the SortedSet has a non-null Comparator.

A problem with mapping *to* Java types *from* Open types is signaled with an [InvalidObjectException](http://docs.google.com/java/io/InvalidObjectException.html). This can happen when an MXBean interface is being analyzed, for example if it references a type that is not *reconstructible* according to the rules above, in a context where a reconstructible type is required. Or it can happen when an instance is being converted (a parameter to a method in an MXBean or a return value from a method in an MXBean proxy), for example from a String to an Enum if there is no Enum constant with that name.

Depending on the context, the OpenDataException or InvalidObjectException may be wrapped in another exception such as [RuntimeMBeanException](http://docs.google.com/javax/management/RuntimeMBeanException.html) or [UndeclaredThrowableException](http://docs.google.com/java/lang/reflect/UndeclaredThrowableException.html). For every thrown exception, the condition *C* will be true: "*e* is OpenDataException or InvalidObjectException (as appropriate), or *C* is true of *e*.[getCause()](http://docs.google.com/java/lang/Throwable.html#getCause())".

**Since:** 1.6

| **Optional Element Summary** | |
| --- | --- |
| boolean | [**value**](http://docs.google.com/javax/management/MXBean.html#value())            True if the annotated interface is an MXBean interface. |

### value

public abstract boolean **value**

True if the annotated interface is an MXBean interface.

**Returns:**true if the annotated interface is an MXBean interface. **Default:**true

| | [**Overview**](http://docs.google.com/overview-summary.html) | [**Package**](http://docs.google.com/package-summary.html) | **Class** | [**Use**](http://docs.google.com/class-use/MXBean.html) | [**Tree**](http://docs.google.com/package-tree.html) | [**Deprecated**](http://docs.google.com/deprecated-list.html) | [**Index**](http://docs.google.com/index-files/index-1.html) | [**Help**](http://docs.google.com/help-doc.html) | | --- | --- | --- | --- | --- | --- | --- | --- | | | ***Java™ Platform***  ***Standard Ed. 6*** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**PREV CLASS**](http://docs.google.com/javax/management/MBeanTrustPermission.html)   [**NEXT CLASS**](http://docs.google.com/javax/management/NotCompliantMBeanException.html) | [**FRAMES**](http://docs.google.com/index.html?javax/management/MXBean.html)    [**NO FRAMES**](http://docs.google.com/MXBean.html)     [**All Classes**](http://docs.google.com/allclasses-noframe.html) |
| SUMMARY: REQUIRED | [OPTIONAL](#17dp8vu) | DETAIL: [ELEMENT](#1ksv4uv) |

[Submit a bug or feature](http://bugs.sun.com/services/bugreport/index.jsp)

For further API reference and developer documentation, see [Java SE Developer Documentation](http://docs.google.com/webnotes/devdocs-vs-specs.html). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2006 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](http://docs.google.com/legal/license.html). Also see the [documentation redistribution policy](http://java.sun.com/docs/redist.html).