



## CHAPTER 9

# *java.beans and java.beans.beancontext*

This chapter covers the `java.beans` and `java.beans.beancontext` packages. `java.beans` defines core classes and interfaces for the JavaBeans component framework. `java.beans.beancontext` defines JavaBeans containers.

### Package `java.beans`

Java 1.1

The `java.beans` package contains classes and interfaces related to JavaBeans components. Most of the classes and interfaces are used by tools that manipulate beans, rather than by the beans themselves. They are also used or implemented by auxiliary classes provided by bean implementors for the benefit of bean-manipulation tools.

The `Beans` class defines several generally useful static methods. Its `instantiate()` method is particularly important. The `Introspector` class is used to obtain information about a bean and the properties, events, and methods it exports. Most of this information is returned using the `FeatureDescriptor` class and its various subclasses. The `java.beans` package also defines the `PropertyChangeEvent` class and the `PropertyChangeListener` interface that are widely used by AWT and Swing to provide notification when a bound property of a GUI component changes.

Java 1.4 adds several new classes to support the JavaBeans persistence mechanism. See `XMLEncoder` for details.

See Chapter 6, for a complete introduction to the JavaBeans component model.

#### *Interfaces:*

```
public interface AppletInitializer;  
public interface BeanInfo;  
public interface Customizer;  
public interface DesignMode;  
public interface ExceptionListener;  
public interface PropertyEditor;  
public interface Visibility;
```

*Events:*

```
public class PropertyChangeEvent extends java.util.EventObject;
```

*Event Listeners:*

```
public interface PropertyChangeListener extends java.util.EventListener;
public interface VetoableChangeListener extends java.util.EventListener;
```

*Other Classes:*

```
public class Beans;
public class Encoder;
    public class XMLEncoder extends Encoder;
public class EventHandler implements java.lang.reflect.InvocationHandler;
public class FeatureDescriptor;
    public class BeanDescriptor extends FeatureDescriptor;
    public class EventSetDescriptor extends FeatureDescriptor;
    public class MethodDescriptor extends FeatureDescriptor;
    public class ParameterDescriptor extends FeatureDescriptor;
    public class PropertyDescriptor extends FeatureDescriptor;
        public class IndexedPropertyDescriptor extends PropertyDescriptor;
public class Introspector;
public abstract class PersistenceDelegate;
    public class DefaultPersistenceDelegate extends PersistenceDelegate;
public class PropertyChangeListenerProxy extends java.util.EventListenerProxy
    implements PropertyChangeListener;
public class PropertyChangeSupport implements Serializable;
public class PropertyEditorManager;
public class PropertyEditorSupport implements PropertyEditor;
public class SimpleBeanInfo implements BeanInfo;
public class Statement;
    public class Expression extends Statement;
public class VetoableChangeListenerProxy extends java.util.EventListenerProxy
    implements VetoableChangeListener;
public class VetoableChangeSupport implements Serializable;
public class XMLDecoder;
```

*Exceptions:*

```
public class IntrospectionException extends Exception;
public class PropertyVetoException extends Exception;
```

**AppletInitializer****Java 1.2****java.beans**

This interface defines general methods to initialize a newly instantiated Applet object. An AppletInitializer can be passed to the Beans.instantiate() method so that when a bean that is also an applet is created, it can be properly initialized. The initialize() method should associate the applet object with an appropriate AppletContext and AppletStub, place it within an appropriate Container, and call its init() method. The activate() method should make the applet active by calling its start() method. This interface is typically used by bean context implementors. Applications writers may need to use AppletInitializer objects, but should not usually have to invoke or implement the methods directly.

```
public interface AppletInitializer {
    // Public Instance Methods
```

## AppletInitializer

```
public abstract void activate(java.applet.Applet newApplet);
public abstract void initialize(java.applet.Applet newAppletBean, java.beans.beancontext.BeanContext bCtxt);
}
```

*Passed To:* Beans.instantiate()

## BeanDescriptor

Java 1.1

java.beans

A **BeanDescriptor** object is a type of **FeatureDescriptor** that describes a JavaBeans component. The **BeanInfo** class for a bean optionally creates and initializes a **BeanDescriptor** object to describe the bean. Typically, only application builders and similar tools use the **BeanDescriptor**. To create a **BeanDescriptor**, you must specify the class of the bean and, optionally, the class of a **Customizer** for the bean. You can use the methods of **FeatureDescriptor** to provide additional information about the bean.



```
public class BeanDescriptor extends FeatureDescriptor {
// Public Constructors
    public BeanDescriptor(Class beanClass);
    public BeanDescriptor(Class beanClass, Class customizerClass);
// Public Instance Methods
    public Class getBeanClass();
    public Class getCustomizerClass();
}
```

*Returned By:* BeanInfo.getBeanDescriptor(), SimpleBeanInfo.getBeanDescriptor()

## BeanInfo

Java 1.1

java.beans

The **BeanInfo** interface defines the methods a class must implement in order to export information about a JavaBeans component. The **Introspector** class knows how to obtain all the basic information required about a bean. A bean that wants to be more programmer-friendly can provide a class that implements this interface, and provide additional information about itself (such as an icon and description strings for each of its properties, events, and methods). Note that a bean developer defines a class that implements the methods of this interface. Typically, only builder applications and similar tools actually invoke the methods defined here.

The **getBeanDescriptor()**, **getEventSetDescriptors()**, **getPropertyDescriptors()**, and **getMethodDescriptors()** methods should return appropriate descriptor objects for the bean or null if the bean does not provide explicit bean, event set, property, or method descriptor objects. The **getDefaultEventIndex()** and **getDefaultPropertyIndex()** methods return values that specify the default event and property (i.e., those most likely to be of interest to a programmer using the bean). These methods should return -1 if there are no defaults. The **getIcon()** method should return an image object suitable for representing the bean in a palette or menu of available beans. The argument passed to this method is one of the four constants defined by the class; it specifies the type and size of icon requested. If the requested icon cannot be provided, **getIcon()** should return null.

A **BeanInfo** class is allowed to return null or -1 if it cannot provide the requested information. In this case, the **Introspector** class provides basic values for the omitted information from its own introspection of the bean. See **SimpleBeanInfo** for a trivial implementation of this interface suitable for convenient subclassing.

```

public interface BeanInfo {
    // Public Constants
    public static final int ICON_COLOR_16x16;           =1
    public static final int ICON_COLOR_32x32;           =2
    public static final int ICON_MONO_16x16;            =3
    public static final int ICON_MONO_32x32;            =4
    // Property Accessor Methods (by property name)
    public abstract BeanInfo[] getAdditionalBeanInfo();
    public abstract BeanDescriptor getBeanDescriptor();
    public abstract int getDefaultEventIndex();
    public abstract int getDefaultPropertyIndex();
    public abstract EventSetDescriptor[] getEventSetDescriptors();
    public abstract MethodDescriptor[] getMethodDescriptors();
    public abstract PropertyDescriptor[] getPropertyDescriptors();
    // Public Instance Methods
    public abstract java.awt.Image getIcon(int iconKind);
}

```

**Implementations:** SimpleBeanInfo, java.beans.beancontext.BeanContextServiceProviderBeanInfo

**Returned By:** BeanInfo.getAdditionalBeanInfo(), Introspector.getBeanInfo(),  
SimpleBeanInfo.getAdditionalBeanInfo(),  
java.beans.beancontext.BeanContextServiceProviderBeanInfo.getServicesBeanInfo()

## Beans

Java 1.1

### java.beans

The Beans class is not meant to be instantiated; its static methods provide miscellaneous JavaBeans features. The `instantiate()` method creates an instance of a bean. The specified bean name represents either a serialized bean file or a bean class file; it is interpreted relative to the specified `ClassLoader` object.

The `setDesignTime()` and `isDesignTime()` methods can set and query a flag that indicates whether beans are being used in a application builder environment. Similarly, `setGuiAvailable()` and `isGuiAvailable()` set and query a flag that indicates whether the Java Virtual Machine is running in an environment in which a GUI is available. (Note that untrusted applet code cannot call `setDesignTime()` or `setGuiAvailable()`.)

The `isInstanceOf()` method is a replacement for the Java `instanceof` operator to use with beans. Currently, it behaves like `instanceof`, but in the future it may work with beans that consist of a set of Java objects, each of which provides a different view of a bean. Similarly, the `getInstanceOf()` method is a replacement for the Java cast operator. This method converts a bean to a superclass or interface type, and currently, it behaves like a cast, but in the future, it will be compatible with multiclass beans.

```

public class Beans {
    // Public Constructors
    public Beans();
    // Public Class Methods
    public static Object getInstanceOf(Object bean, Class targetType);
    public static Object instantiate(ClassLoader cls, String beanName) throws java.io.IOException,
        ClassNotFoundException;
    1.2 public static Object instantiate(ClassLoader cls, String beanName,
        java.beans.beancontext.BeanContext beanContext) throws java.io.IOException,
        ClassNotFoundException;
    1.2 public static Object instantiate(ClassLoader cls, String beanName,
        java.beans.beancontext.BeanContext beanContext, AppletInitializer initializer)
        throws java.io.IOException, ClassNotFoundException;
}

```

## Beans

```
public static boolean isDesignTime();  
public static boolean isGuiAvailable();  
public static boolean isInstanceOf(Object bean, Class targetType);  
public static void setDesignTime(boolean isDesignTime) throws SecurityException;  
public static void setGuiAvailable(boolean isGuiAvailable) throws SecurityException;  
}
```

### Customizer

Java 1.1

#### java.beans

The **Customizer** interface specifies the methods that must be defined by any class designed to customize a JavaBeans component. In addition to implementing this interface, a customizer class must be a subclass of **java.awt.Component** and have a constructor that takes no arguments so it can be instantiated by an application builder.

Customizer classes are typically used by a complex bean to allow the user to easily configure the bean and provide an alternative to a simple list of properties and their values. If a customizer class is defined for a bean, it must be associated with the bean through a **BeanDescriptor** object returned by a **BeanInfo** class for the bean. Note that while a **Customizer** class is created by the author of a bean, that class is instantiated and used only by application builders and similar tools.

After a **Customizer** class is instantiated, its **setObject()** method is invoked once to specify the bean object to customize. The **addPropertyChangeListener()** and **removePropertyChangeListener()** methods can be called to register and deregister **PropertyChangeListener** objects. The **Customizer** should send a **PropertyChangeEvent** to all registered listeners any time it changes a property of the bean it is customizing.

```
public interface Customizer {  
    // Event Registration Methods (by event name)  
    public abstract void addPropertyChangeListener(PropertyChangeListener listener);  
    public abstract void removePropertyChangeListener(PropertyChangeListener listener);  
    // Public Instance Methods  
    public abstract void setObject(Object bean);  
}
```

### DefaultPersistenceDelegate

Java 1.4

#### java.beans

This class is the persistence delegate used to encode instances of classes that do not have a built-in delegate, that do not have a delegate specified by their **BeanInfo**, and for which no custom delegate has been registered. The JavaBeans persistence mechanism will use the no-argument constructor to instantiate a **DefaultPersistenceDelegate** as needed for any such class. The resulting persistence delegate assumes that the class has a no-argument constructor, and that its state can be completely described using pairs of property accessor methods. These methods must follow the standard JavaBeans get/set naming convention. When working with conforming JavaBeans classes such as these, there is never any need for an application to instantiate a **DefaultPersistenceDelegate** because this is done automatically by the beans persistence mechanism.

Instances of this class can also serve as a persistence delegate for beans that do not have a no-argument constructor, and that instead pass the values of read-only properties to the constructor. In this case, simply instantiate a **DefaultPersistenceDelegate** by passing an array of strings listing the names of the read-only constructor arguments. The bean class must of course define property getter methods for these named properties. These getter methods must follow the normal JavaBeans naming conventions, or must be specified in a **BeanInfo** class. Consider the **java.awt.Color** class, for example. (**Color** is

documented in *Java Foundation Classes in a Nutshell* [O'Reilly].) If the JavaBeans persistence mechanism did not already define a persistence delegate for this class, you could create one like this:

```
PersistenceDelegate delegate = new DefaultPersistenceDelegate(new String[]{"red", "green", "blue"});
```

This works because `Color` defines a three-argument constructor whose three arguments correspond to the return values of the property getter methods `getRed()`, `getGreen()`, and `getBlue()`.

Object	PersistenceDelegate	DefaultPersistenceDelegate
--------	---------------------	----------------------------

```
public class DefaultPersistenceDelegate extends PersistenceDelegate {
// Public Constructors
    public DefaultPersistenceDelegate();
    public DefaultPersistenceDelegate(String[] constructorPropertyNames);
// Protected Methods Overriding PersistenceDelegate
    protected void initialize(Class type, Object oldInstance, Object newInstance, Encoder out);
    protected Expression instantiate(Object oldInstance, Encoder out);
    protected boolean mutatesTo(Object oldInstance, Object newInstance);
}
```

## DesignMode

Java 1.2

### java.beans

This interface defines a single boolean `designTime` property that specifies whether a bean is running within an interactive design tool or a standalone application or applet. This interface is typically implemented by a bean container or bean context, so that children beans can query the `designTime` property.

```
public interface DesignMode {
// Public Constants
    public static final String PROPERTYNAME;           = [quot ] designTime [quot ]
// Public Instance Methods
    public abstract boolean isDesignTime();
    public abstract void setDesignTime(boolean designTime);
}
```

*Implementations:* `java.beans.beancontext.BeanContext`

## Encoder

Java 1.4

### java.beans

This class provides the architectural underpinnings for the JavaBeans persistence mechanism; the job of an `Encoder` object is to encode and output the `Expression` and `Statement` objects produced by a `PersistenceDelegate` to describe the state of a bean. Although `Encoder` is not technically an `abstract` class, it is never used directly, since it does not perform any useful encoding itself. See the `XMLEncoder` subclass for further details.

```
public class Encoder {
// Public Constructors
    public Encoder();
// Public Instance Methods
    public Object get(Object oldInstance);
    public ExceptionListener getExceptionListener();
    public PersistenceDelegate getPersistenceDelegate(Class type);
    public Object remove(Object oldInstance);
    public void setExceptionListener(ExceptionListener exceptionListener);
}
```

## Encoder

```
public void setPersistenceDelegate(Class type, PersistenceDelegate persistenceDelegate);
public void writeExpression(Expression oldExp);
public void writeStatement(java.beans.Statement oldStm);
// Protected Instance Methods
protected void writeObject(Object o);
}
```

*Subclasses:* XMLEncoder

*Passed To:* DefaultPersistenceDelegate.{initialize(), instantiate()}, PersistenceDelegate.{initialize(), instantiate(), writeObject()}

## EventHandler

Java 1.4

### java.beans

This class uses `Proxy` and other classes from the `java.lang.reflect` package to create objects that can serve as simple event listeners and can be easily serialized using the JavaBeans persistence mechanism in a way that event listeners implemented as anonymous classes cannot.

Applications do not typically work with `EventHandler` objects or the `EventHandler()` constructor directly. Instead, they use one of the static `create()` methods to obtain an object that implements the desired event listener interface. A listener created with `EventHandler.create()` performs a single method call when activated. You specify the name of the method to call, the object upon which the method is to be called, and an optional argument to pass to the method. This argument value may be the event object that triggered the listener, or it may be a property of that event object. The arguments to `create()` are the following:

#### *listenerInterface*

The Class object that represents the `EventListener` to be returned. For example, `PropertyChangeListener.class`.

#### *target*

The object upon which the resulting event listener should invoke a method.

#### *action*

The name of the method to be invoked on *target*. This argument may also specify the name of a property of *target*, in which case a property setter method will be invoked. So instead of specifying the method name "setX()" as the *action* you could also specify the property name "x".

#### *eventPropertyName*

This optional argument specifies the name of a property of the `EventObject` object (or whatever other object is passed to the listener method) whose value is to be passed to the *action* method of the *target* object. If you use the three-argument version of `create()`, or pass null for this argument, then `EventHandler` will either pass the event object itself as the argument to the *action* method, or will pass no argument at all.

If *eventPropertyName* is "x", the `EventHandler` first looks for a no-argument property getter method named `getX()` in the object (typically an `EventObject`) passed to the listener method. If no such method exists, it then looks for a no-argument getter named "isX". If that method does not exist either, it then looks for a no-argument method named "x" and uses that method.

*eventPropertyName* may actually specify multiple levels of property lookups using dot-separated names. For example, a *eventPropertyName* of “propertyName.toLowerCase” would be translated into the calls `getPropertyName().toLowerCase()` on the event object. The value returned by these calls would then be passed to the *action* method.

#### listenerMethodName

This argument specifies the name of the method in the *listenerInterface* which is to be implemented by the returned object. If you specify `null` or use the three- or four-argument version of `create()` then all methods of the returned object will invoke the *action* method of the *target* object.

Object	EventHandler	InvocationHandler
--------	--------------	-------------------

```

public class EventHandler implements java.lang.reflect.InvocationHandler {
// Public Constructors
    public EventHandler(Object target, String action, String eventPropertyName, String listenerMethodName);
// Public Class Methods
    public static Object create(Class listenerInterface, Object target, String action);
    public static Object create(Class listenerInterface, Object target, String action, String eventPropertyName);
    public static Object create(Class listenerInterface, Object target, String action, String eventPropertyName,
                               String listenerMethodName);
// Public Instance Methods
    public String getAction();
    public String getEventPropertyName();
    public String getListenerMethodName();
    public Object getTarget();
// Methods Implementing InvocationHandler
    public Object invoke(Object proxy, java.lang.reflect.Method method, Object[] arguments);
}

```

## EventSetDescriptor

Java 1.1

### java.beans

An `EventSetDescriptor` object is a type of `FeatureDescriptor` that describes a single set of events supported by a JavaBeans component. A set of events corresponds to one or more methods supported by a single `EventListener` interface. The `BeanInfo` class for a bean optionally creates `EventSetDescriptor` objects to describe the event sets the bean supports. Typically, only application builders and similar tools use the `get` and `is` methods of `EventSetDescriptor` objects to obtain the event-set description information.

To create an `EventSetDescriptor` object, you must specify the class of the bean that supports the event set, the base name of the event set, the class of the `EventListener` interface that corresponds to the event set, and the methods within this interface that are invoked when particular events within the set occur. Optionally, you can also specify the methods of the bean class that add and remove `EventListener` objects. The various constructors allow you to specify methods by name, as `java.lang.reflect.Method` objects, or as `MethodDescriptor` objects. In Java 1.4 and later, you can also specify the “get listeners” method used to query the set of registered listener objects for a event set.

Once you have created an `EventSetDescriptor`, use `setUnicast()` to specify whether it represents a unicast event and `setDefaultEventSet()` to specify whether the event set should be treated as the default event set by builder applications. The methods of the `FeatureDescriptor` superclass allow additional information about the property to be specified.

Object	FeatureDescriptor	EventSetDescriptor
--------	-------------------	--------------------



## EventSetDescriptor

```
public class EventSetDescriptor extends FeatureDescriptor {  
    // Public Constructors  
    public EventSetDescriptor(Class sourceClass, String eventSetName, Class listenerType,  
        String listenerMethodName) throws IntrospectionException;  
    public EventSetDescriptor(String eventSetName, Class listenerType, java.lang.reflect.Method[] listenerMethods,  
        java.lang.reflect.Method addListenerMethod,  
        java.lang.reflect.Method removeListenerMethod) throws IntrospectionException;  
    public EventSetDescriptor(String eventSetName, Class listenerType,  
        MethodDescriptor[] listenerMethodDescriptors,  
        java.lang.reflect.Method addListenerMethod,  
        java.lang.reflect.Method removeListenerMethod) throws IntrospectionException;  
    1.4 public EventSetDescriptor(String eventSetName, Class listenerType, java.lang.reflect.Method[] listenerMethods,  
        java.lang.reflect.Method addListenerMethod,  
        java.lang.reflect.Method removeListenerMethod,  
        java.lang.reflect.Method getListenerMethod) throws IntrospectionException;  
    public EventSetDescriptor(Class sourceClass, String eventSetName, Class listenerType,  
        String[] listenerMethodNames, String addListenerMethodName,  
        String removeListenerMethodName) throws IntrospectionException;  
    1.4 public EventSetDescriptor(Class sourceClass, String eventSetName, Class listenerType,  
        String[] listenerMethodNames, String addListenerMethodName,  
        String removeListenerMethodName, String getListenerMethodName)  
        throws IntrospectionException;  
    // Property Accessor Methods (by property name)  
    public java.lang.reflect.Method getAddListenerMethod();  
    1.4 public java.lang.reflect.Method getGetListenerMethod();  
    public boolean isInDefaultEventSet();  
    public void setInDefaultEventSet(boolean inDefaultEventSet);  
    public MethodDescriptor[] getListenerMethodDescriptors();  
    public java.lang.reflect.Method[] getListenerMethods();  
    public Class getListenerType();  
    public java.lang.reflect.Method getRemoveListenerMethod();  
    public boolean isUnicast();  
    public void setUnicast(boolean unicast);  
}
```

*Returned By:* BeanInfo.getEventSetDescriptors(), SimpleBeanInfo.getEventSetDescriptors()

## ExceptionListener

Java 1.4

### java.beans

This interface is implemented by objects that wish to be notified of exceptions that occur during the JavaBeans encoding or decoding process performed by the `Encoder` or `XMLDecoder` classes. The `exceptionThrown()` method will be invoked when an exception occurs.

```
public interface ExceptionListener {  
    // Public Instance Methods  
    public abstract void exceptionThrown(Exception e);  
}
```

*Passed To:* `Encoder.setExceptionListener()`, `XMLDecoder.{setExceptionListener(), XMLDecoder()}`

*Returned By:* `Encoder.getExceptionListener()`, `XMLDecoder.getExceptionListener()`

**Expression**

Java 1.4

**java.beans**

This subclass of **Statement** represents a method call that has a return value. You do not use the inherited `execute()` method to invoke an **Expression**, but instead call `getValue()` to invoke the described method and return its return value. `getValue()` caches the resulting value, so that only the first call to `getValue()` actually results in an invocation of the underlying method. You can also specify the value by passing it to the four-argument constructor or to `setValue()`. This is sometimes done for efficiency when encoding a method call whose result you already know. See **Statement** for information about specifying the target object, the method name, and the array of method argument objects.

There are two special-case method names you can use with the **Expression** class. To invoke a **constructor**, use the **Class** object for the desired class as the target object, and use "new" as the method name. To access an element of an array, specify the array as the target object, use "get" as the method name, and pass an **Integer** containing the desired array index as the sole method argument.



```

public class Expression extends java.beans.Statement {
// Public Constructors
    public Expression(Object target, String methodName, Object[ ] arguments);
    public Expression(Object value, Object target, String methodName, Object[ ] arguments);
// Public Instance Methods
    public Object getValue() throws Exception;
    public void setValue(Object value);
// Public Methods Overriding Statement
    public String toString();
}
  
```

*Passed To:* `Encoder.writeExpression()`, `XMLEncoder.writeExpression()`

*Returned By:* `DefaultPersistenceDelegate.instantiate()`, `PersistenceDelegate.instantiate()`

**FeatureDescriptor**

Java 1.1

**java.beans**

The **FeatureDescriptor** class is the base class for **MethodDescriptor** and **PropertyDescriptor**, as well as other classes used by the JavaBeans introspection mechanism. It provides basic information about a feature (e.g., method, property, or event) of a bean. Typically, the methods that begin with `get` and `is` are used by application builders or other tools to query the features of a bean. The `set` methods, on the other hand, may be used by bean authors to define information about the bean.

`setName()` specifies the locale-independent, programmatic name of the feature; `setDisplayName()` specifies a localized, human-readable name; and `setShortDescription()` specifies a short localized string (about 40 characters) that describes the feature. Both the short description and the localized name default to the value of the programmatic name. `setExpert()` and `setHidden()` allow you to indicate that the feature is for use only by experts or by the builder tool and should be hidden from users of the builder. Finally, the `setValue()` method allows you to associate an arbitrary named value with the feature.

```

public class FeatureDescriptor {
// Public Constructors
    public FeatureDescriptor();
// Property Accessor Methods (by property name)
    public String getDisplayName(); default:null
}
  
```

## FeatureDescriptor

```
public void setDisplayDisplayName(String displayName);
public boolean isExpert(); default:false
public void setExpert(boolean expert);
public boolean isHidden(); default:false
public void setHidden(boolean hidden);
public String getName(); default:null
public void setName(String name);
1.2 public boolean isPreferred(); default:false
1.2 public void setPreferred(boolean preferred);
public String getShortDescription(); default:null
public void setShortDescription(String text);
// Public Instance Methods
public java.util.Enumeration attributeNames();
public Object getValue(String attributeName);
public void setValue(String attributeName, Object value);
}
```

**Subclasses:** BeanDescriptor, EventSetDescriptor, MethodDescriptor, ParameterDescriptor, PropertyDescriptor

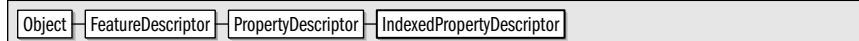
## IndexedPropertyDescriptor

Java 1.1

### java.beans

An `IndexedPropertyDescriptor` object is a type of `PropertyDescriptor` that describes a bean property that is (or behaves like) an array. The `BeanInfo` class for a bean optionally creates and initializes `IndexedPropertyDescriptor` objects to describe the indexed properties the bean supports. Typically, only application builders and similar tools use the descriptor objects to obtain indexed property description information.

You create an `IndexedPropertyDescriptor` by specifying the name of the indexed property and the `Class` object for the bean. If you have not followed the standard design patterns for accessor method naming, you can also specify the accessor methods for the property, either as method names or as `java.lang.reflect.Method` objects. Once you have created an `IndexedPropertyDescriptor` object, you can use the methods of `PropertyDescriptor` and `FeatureDescriptor` to provide additional information about the indexed property.



```
public class IndexedPropertyDescriptor extends PropertyDescriptor {
// Public Constructors
public IndexedPropertyDescriptor(String propertyName, Class beanClass) throws IntrospectionException;
public IndexedPropertyDescriptor(String propertyName, java.lang.reflect.Method getter,
    java.lang.reflect.Method setter, java.lang.reflect.Method indexedGetter,
    java.lang.reflect.Method indexedSetter) throws IntrospectionException;
public IndexedPropertyDescriptor(String propertyName, Class beanClass, String getterName,
    String setterName, String indexedGetterName, String indexedSetterName)
    throws IntrospectionException;
// Public Instance Methods
public Class getIndexedPropertyType();
public java.lang.reflect.Method getIndexedReadMethod();
public java.lang.reflect.Method getIndexedWriteMethod();
1.2 public void setIndexedReadMethod(java.lang.reflect.Method getter) throws IntrospectionException;
1.2 public void setIndexedWriteMethod(java.lang.reflect.Method setter) throws IntrospectionException;
// Public Methods Overriding PropertyDescriptor
1.4 public boolean equals(Object obj);
}
```

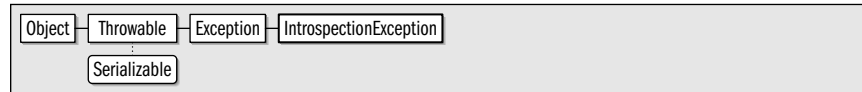
**IntrospectionException**

Java 1.1

java.beans

serializable checked

An `IntrospectionException` signals that introspection on a JavaBeans component cannot be completed. Typically, this indicates a bug in the way the bean or its associated `BeanInfo` class is defined.



```

public class IntrospectionException extends Exception {
// Public Constructors
    public IntrospectionException(String mess);
}
  
```

*Thrown By:* Too many methods to list.

**Introspector**

Java 1.1

java.beans

The `Introspector` is a class that is never instantiated. Its static `getBeanInfo()` methods provide a way to obtain information about a JavaBeans component and are typically only invoked by application builders or similar tools. `getBeanInfo()` first looks for a `BeanInfo` class for the specified bean class. For a class named *x*, it looks for a `BeanInfo` class named *xBeanInfo*, first in the current package and then in each of the packages in the `BeanInfo` search path.

If no `BeanInfo` class is found, or if the `BeanInfo` class found does not provide complete information about the bean properties, events, and methods, `getBeanInfo()` introspects on the bean class by using the `java.lang.reflect` package to fill in the missing information. When explicit information is provided by a `BeanInfo` class, `getBeanInfo()` treats it as definitive. When determining information through introspection, however, it examines each of the bean's superclasses in turn, looking for a `BeanInfo` class at that level or using introspection. When calling `getBeanInfo()`, you may optionally specify a second class argument that specifies a superclass for which, and above which, `getBeanInfo()` does not introspect.

```

public class Introspector {
// No Constructor
// Public Constants
    1.2 public static final int IGNORE_ALL_BEANINFO;           =3
    1.2 public static final int IGNORE_IMMEDIATE_BEANINFO;   =2
    1.2 public static final int USE_ALL_BEANINFO;             =1
// Public Class Methods
    public static String decapitalize(String name);
    1.2 public static void flushCaches();
    1.2 public static void flushFromCaches(Class clz);
        public static BeanInfo getBeanInfo(Class beanClass) throws IntrospectionException;
    1.2 public static BeanInfo getBeanInfo(Class beanClass, int flags) throws IntrospectionException;
        public static BeanInfo getBeanInfo(Class beanClass, Class stopClass) throws IntrospectionException;
        public static String[] getBeanInfoSearchPath();           synchronized
        public static void setBeanInfoSearchPath(String[] path);   synchronized
}
  
```

**MethodDescriptor**

Java 1.1

**java.beans**

A **MethodDescriptor** object is a type of **FeatureDescriptor** that describes a method supported by a JavaBeans component. The **BeanInfo** class for a bean optionally creates **MethodDescriptor** objects that describe the methods the bean exports. While a **BeanInfo** class creates and initializes **MethodDescriptor** objects, it is typically only application builders and similar tools that use these objects to obtain information about the methods supported by a bean.

To create a **MethodDescriptor**, you must specify the `java.lang.reflect.Method` object for the method and, optionally, an array of **ParameterDescriptor** objects that describe the parameters of the method. Once you have created a **MethodDescriptor** object, you can use **FeatureDescriptor** methods to provide additional information about each method.



```

public class MethodDescriptor extends FeatureDescriptor {
// Public Constructors
    public MethodDescriptor(java.lang.reflect.Method method);
    public MethodDescriptor(java.lang.reflect.Method method, ParameterDescriptor[ ] parameterDescriptors);
// Public Instance Methods
    public java.lang.reflect.Method getMethod();
    public ParameterDescriptor[ ] getParameterDescriptors();
}
  
```

*Passed To:* EventSetDescriptor.EventSetDescriptor()

*Returned By:* BeanInfo.getMethodDescriptors(), EventSetDescriptor.getListenerMethodDescriptors(), SimpleBeanInfo.getMethodDescriptors()

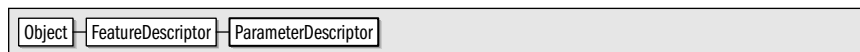
**ParameterDescriptor**

Java 1.1

**java.beans**

A **ParameterDescriptor** object is a type of **FeatureDescriptor** that describes an argument or parameter to a method of a JavaBeans component. The **BeanInfo** class for a JavaBeans component optionally creates **ParameterDescriptor** objects that describe the parameters of the methods the bean exports. While the **BeanInfo** class creates and initializes **ParameterDescriptor** objects, it is typically only application builders and similar tools that use these objects to obtain information about method parameters supported by the bean.

The **ParameterDescriptor** class is a trivial subclass of **FeatureDescriptor** and does not provide any new methods. Thus, you should use the methods of **FeatureDescriptor** to provide information about method parameters.



```

public class ParameterDescriptor extends FeatureDescriptor {
// Public Constructors
    public ParameterDescriptor();
}
  
```

*Passed To:* MethodDescriptor.MethodDescriptor()

*Returned By:* MethodDescriptor.getParameterDescriptors()

**PersistenceDelegate**

Java 1.4

**java.beans**

A **PersistenceDelegate** plays the central role in the JavaBeans persistence mechanism: it is responsible for determining the current state of a bean, and expressing that state in terms of the public API of the bean as a series of constructor and method calls (represented by **Statement** and **Expression** objects) that can serve to later recreate the bean and its state.

**writeObject()** is the key method: an **Encoder** passes an object to the **writeObject()** method of the appropriate **PersistenceDelegate**. It is the delegate's responsibility to determine the sequence of constructor and method calls required to recreate the object, and to tell the **Encoder** what those methods calls are by passing **Statement** and **Expression** objects to the encoder's **writeStatement()** and **writeExpression()** methods. The **Encoder** can then encode the **Statement** and **Expression** objects into some output format, such as an XML document. Note that **writeObject()** is the only public method of the **PersistenceDelegate** class. It is implemented in terms of the three protected methods, which subclasses can override to create custom persistence behavior.

The JavaBeans persistence implementation provides working delegates for all AWT and Swing GUI components, and for all types (such as colors and fonts) used as properties of those components. The **DefaultPersistenceDelegate** class works for any bean that expresses its complete state in terms of constructor arguments and property getter and setter methods. If you want to use the JavaBeans persistence mechanism to serialize objects that do not conform to these JavaBeans design conventions, you may need to subclass **PersistenceDelegate** and override one or more of its three protected methods. A full tutorial on writing custom persistence delegates is beyond the scope of this book, but briefly, the **instantiate()** method is responsible for returning an **Expression** object that represents a call to a constructor or factory method to instantiate an object with the same state as the specified object. **mutatesTo()** is supposed to determine whether one instance of an object can be mutated so that it has the same state as another object. If this method returns true, then the **PersistenceDelegate** may call the **initialize()** method which must actually write out the **Expression** and **Statement** objects that represent the method calls required to perform that initialization.

If you need to use an instance of **DefaultPersistenceDelegate** or an instance of a custom **PersistenceDelegate** you have written yourself in order to persistently save the state of a bean or related object, you must first associate the delegate with the class for which it is responsible by calling the **setPersistenceDelegate()** method of your **XMLEncoder** object. (Note that this method is not defined by **XMLEncoder**, but is actually inherited from the **Encoder** superclass.) Alternatively, a **JavaBean** may specify its **PersistenceDelegate** by defining a **BeanInfo** class that returns a **BeanDescriptor** that returns the name of the persistence delegate class as the value of the attribute returned by **getValue("persistenceDelegate")**.

```
public abstract class PersistenceDelegate {
    // Public Constructors
    public PersistenceDelegate();
    // Public Instance Methods
    public void writeObject(Object oldInstance, Encoder out);
    // Protected Instance Methods
    protected void initialize(Class type, Object oldInstance, Object newInstance, Encoder out);
    protected abstract Expression instantiate(Object oldInstance, Encoder out);
    protected boolean mutatesTo(Object oldInstance, Object newInstance);
}
```

## *PersistenceDelegate*

*Subclasses:* DefaultPersistenceDelegate

*Passed To:* Encoder.setPersistenceDelegate()

*Returned By:* Encoder.getPersistenceDelegate()

## PropertyChangeEvent

Java 1.1

java.beans

serializable event

PropertyChangeEvent is a subclass of java.util.EventObject. An event of this type is sent to interested PropertyChangeListener objects whenever a JavaBeans component changes a bound property or whenever a PropertyEditor or Customizer changes a property value. A PropertyChangeEvent is also sent to registered VetoableChangeListener objects when a bean attempts to change the value of a constrained property.

When creating a PropertyChangeEvent, you normally specify the bean that generated the event, the programmatic (locale-independent) name of the property that changed, and the old and new values of the property. If the values cannot be determined, null should be passed instead. If the event is a notification that more than one property value changed, the name should also be null. While JavaBeans must generate and send PropertyChangeEvent objects, it is typically only application builders and similar tools that are interested in receiving them.



```
public class PropertyChangeEvent extends java.util.EventObject {
// Public Constructors
    public PropertyChangeEvent(Object source, String propertyName, Object oldValue, Object newValue);
// Public Instance Methods
    public Object getNewValue();
    public Object getOldValue();
    public Object getPropagationId();
    public String getPropertyName();
    public void setPropagationId(Object propagationId);
}
```

*Passed To:* Too many methods to list.

*Returned By:* PropertyVetoException.getPropertyChangeEvent()

## PropertyChangeListener

Java 1.1

java.beans

event listener

This interface is an extension of java.util.EventListener; it defines the method a class must implement in order to be notified when property changes occur. A PropertyChangeEvent is sent to all registered PropertyChangeListener objects when a bean changes one of its bound properties or when a PropertyEditor or Customizer changes the value of a property.



```
public interface PropertyChangeListener extends java.util.EventListener {
// Public Instance Methods
    public abstract void propertyChange(PropertyChangeEvent evt);
}
```

**Implementations:** Too many classes to list.

**Passed To:** Too many methods to list.

**Returned By:** Too many methods to list.

**Type Of:** javax.swing.plaf.basic.BasicColorChooserUI.propertyChangeListener,  
 javax.swing.plaf.basic.BasicComboBoxUI.propertyChangeListener,  
 javax.swing.plaf.basic.BasicComboPopup.propertyChangeListener,  
 javax.swing.plaf.basic.BasicInternalFrameTitlePane.propertyChangeListener,  
 javax.swing.plaf.basic.BasicInternalFrameUI.propertyChangeListener,  
 javax.swing.plaf.basic.BasicListUI.propertyChangeListener,  
 javax.swing.plaf.basic.BasicMenuUI.propertyChangeListener,  
 javax.swing.plaf.basic.BasicOptionPaneUI.propertyChangeListener,  
 javax.swing.plaf.basic.BasicScrollBarUI.propertyChangeListener,  
 javax.swing.plaf.basic.BasicScrollPaneUI.spPropertyChangeListener,  
 javax.swing.plaf.basic.BasicSliderUI.propertyChangeListener,  
 javax.swing.plaf.basic.BasicSplitPaneUI.propertyChangeListener,  
 javax.swing.plaf.basic.BasicTabbedPaneUI.propertyChangeListener,  
 javax.swing.plaf.basic.BasicToolBarUI.propertyListener,  
 javax.swing.plaf.metal.MetalToolBarUI.rolloverListener

## PropertyChangeListenerProxy

Java 1.4

java.beans

This class implements `PropertyChangeListener` and serves as a wrapper around another `PropertyChangeListener` object. Beans that implement a two-argument `addPropertyChangeListener()` method to allow registration of a property change listener for a specific named property may return instances of this class from their `getPropertyChangeListeners()` method (if they implement one). Use `getPropertyName()` to determine the name of the property to which the listener applies. And use the inherited `getListener()` method to obtain the underlying `PropertyChangeListener` object.



```

public class PropertyChangeListenerProxy extends java.util.EventListenerProxy
    implements PropertyChangeListener {
// Public Constructors
    public PropertyChangeListenerProxy(String propertyName, PropertyChangeListener listener);
// Public Instance Methods
    public String getPropertyName();
// Methods Implementing PropertyChangeListener
    public void propertyChange(PropertyChangeEvent evt);
}
    
```

## PropertyChangeSupport

Java 1.1

java.beans

serializable

The `PropertyChangeSupport` class is a convenience class that maintains a list of registered `PropertyChangeListener` objects and provides the `firePropertyChange()` method for sending a `PropertyChangeEvent` object to all registered listeners. Because there are some tricky thread-synchronization issues involved in doing this correctly, it is recommended that all JavaBeans that support bound properties either extend this class or, more commonly, create an instance of this class to which they can delegate their `addProperty-`



## PropertyChangeSupport

ChangeListener() and removePropertyChangeListener() methods. In Java 1.4, beans that define getPropertyChangeListeners() methods can also delegate that method to this object.

```
Object | PropertyChangeSupport | Serializable

public class PropertyChangeSupport implements Serializable {
// Public Constructors
    public PropertyChangeSupport(Object sourceBean);
// Event Registration Methods (by event name)
    public void addPropertyChangeListener(PropertyChangeListener listener);           synchronized
    public void removePropertyChangeListener(PropertyChangeListener listener);       synchronized
// Public Instance Methods
    1.2 public void addPropertyChangeListener(String propertyName, PropertyChangeListener listener);   synchronized
    1.2 public void firePropertyChange(PropertyChangeEvent evt);
    1.2 public void firePropertyChange(String propertyName, int oldValue, int newValue);
    1.2 public void firePropertyChange(String propertyName, boolean oldValue, boolean newValue);
    public void firePropertyChange(String propertyName, Object oldValue, Object newValue);
    1.4 public PropertyChangeListener[] getPropertyChangeListeners();           synchronized
    1.4 public PropertyChangeListener[] getPropertyChangeListeners(String propertyName);   synchronized
    1.2 public boolean hasListeners(String propertyName);           synchronized
    1.2 public void removePropertyChangeListener(String propertyName,
                                                PropertyChangeListener listener);
}
```

*Subclasses:* javax.swing.event.SwingPropertyChangeSupport

*Type Of:* java.awt.Toolkit.desktopPropsSupport,  
java.beans.beancontext.BeanContextChildSupport.pcSupport

## PropertyDescriptor

Java 1.1

java.beans

A **PropertyDescriptor** object is a type of **FeatureDescriptor** that describes a single property of a JavaBeans component. The **BeanInfo** class for a bean optionally creates and initializes **PropertyDescriptor** objects to describe the properties the bean supports. Typically, only application builders and similar tools use the **get** and **is** methods to obtain this property description information.

You create a **PropertyDescriptor** by specifying the name of the property and the **Class** object for the bean. If you have not followed the standard design patterns for accessor-method naming, you can also specify the accessor methods for the property. Once a **PropertyDescriptor** is created, the **setBound()** and **setConstrained()** methods allow you to specify whether the property is bound and/or constrained. **setPropertyEditorClass()** allows you to specify a specific property editor that should edit the value of this property (this is useful, for example, when the property is an enumerated type with a specific list of supported values). The methods of the **FeatureDescriptor** superclass allow additional information about the property to be specified.

```
Object | FeatureDescriptor | PropertyDescriptor

public class PropertyDescriptor extends FeatureDescriptor {
// Public Constructors
    public PropertyDescriptor(String propertyName, Class beanClass) throws IntrospectionException;
    public PropertyDescriptor(String propertyName, java.lang.reflect.Method getter, java.lang.reflect.Method setter)
        throws IntrospectionException;
    public PropertyDescriptor(String propertyName, Class beanClass, String getterName, String setterName)
        throws IntrospectionException;
```

```
// Property Accessor Methods (by property name)
public boolean isBound();
public void setBound(boolean bound);
public boolean isConstrained();
public void setConstrained(boolean constrained);
public Class getPropertyEditorClass();
public void setPropertyEditorClass(Class propertyEditorClass);
public Class getPropertyType();
public java.lang.reflect.Method getReadMethod();
1.2 public void setReadMethod(java.lang.reflect.Method getter) throws IntrospectionException;
public java.lang.reflect.Method getWriteMethod();
1.2 public void setWriteMethod(java.lang.reflect.Method setter) throws IntrospectionException;
// Public Methods Overriding Object
1.4 public boolean equals(Object obj);
}
```

*Subclasses:* IndexedPropertyDescriptor

*Returned By:* BeanInfo.getPropertyDescriptors(), SimpleBeanInfo.getPropertyDescriptors()

## PropertyEditor

Java 1.1

### java.beans

The `PropertyEditor` interface defines the methods that must be implemented by a JavaBeans property editor intended for use within an application builder or similar tool. `PropertyEditor` is a complex interface because it defines methods to support different ways of displaying property values to the user. It also defines methods to support different ways of allowing the user to edit the property value.

For a property of type *x*, the author of a bean typically implements a property editor of class *xEditor*. While the editor is implemented by the bean author, it is usually instantiated or used only by application builders or similar tools (or by a `Customizer` class for a bean). In addition to implementing the `PropertyEditor` interface, a property editor must have a constructor that expects no arguments, so that it can be easily be instantiated by an application builder. Also, it must accept registration and deregistration of `PropertyChangeListener` objects and send a `PropertyChangeEvent` to all registered listeners when it changes the value of the property being edited. The `PropertyEditorSupport` class is a trivial implementation of `PropertyEditor`, suitable for subclassing or for supporting a list of `PropertyChangeListener` objects.

```
public interface PropertyEditor {
// Event Registration Methods (by event name)
public abstract void addPropertyChangeListener(PropertyChangeListener listener);
public abstract void removePropertyChangeListener(PropertyChangeListener listener);
// Property Accessor Methods (by property name)
public abstract String getAsText();
public abstract void setAsText(String text) throws IllegalArgumentException;
public abstract java.awt.Component getCustomEditor();
public abstract String getJavaInitializationString();
public abstract boolean isPaintable();
public abstract String[] getTags();
public abstract Object getValue();
public abstract void setValue(Object value);
// Public Instance Methods
public abstract void paintValue(java.awt.Graphics gfx, java.awt.Rectangle box);
}
```

## PropertyEditor

```
public abstract boolean supportsCustomEditor();  
}
```

*Implementations:* PropertyEditorSupport

*Returned By:* PropertyEditorManager.findEditor()

## PropertyEditorManager

Java 1.1

java.beans

The PropertyEditorManager class is not meant to be instantiated; it defines static methods for registering and looking up PropertyEditor classes for a specified property type. A bean can specify a particular PropertyEditor class for a given property by specifying it in a PropertyDescriptor object for the property. If it does not do this, the PropertyEditorManager is used to register and look up editors. A bean or an application builder tool can call the registerEditor() method to register a PropertyEditor for properties of a specified type. Application builders and bean Customizer classes can call the findEditor() method to obtain a PropertyEditor for a given property type. If no editor has been registered for a given type, the PropertyEditorManager attempts to locate one. For a type x, it looks for a class xEditor first in the same package as x, and then in each package listed in the property editor search path.

```
public class PropertyEditorManager {  
    // Public Constructors  
    public PropertyEditorManager();  
    // Public Class Methods  
    public static PropertyEditor findEditor(Class targetType); synchronized  
    public static String[] getEditorSearchPath(); synchronized  
    public static void registerEditor(Class targetType, Class editorClass);  
    public static void setEditorSearchPath(String[] path); synchronized  
}
```

## PropertyEditorSupport

Java 1.1

java.beans

The PropertyEditorSupport class is a trivial implementation of the PropertyEditor interface. It provides no-op default implementations of most methods, so you can define simple PropertyEditor subclasses that override only a few required methods. In addition, PropertyEditorSupport defines working versions of addPropertyChangeListener() and removePropertyChangeListener(), along with a firePropertyChange() method that sends a PropertyChangeEvent to all registered listeners. PropertyEditor classes may choose to instantiate a PropertyEditorSupport object simply to handle the job of managing the list of listeners. When used in this way, the PropertyEditorSupport object should be instantiated with a source object specified, so that the source object can be used in the PropertyChangeEvent objects that are sent.

```
graph LR  
    Object --> PropertyEditorSupport  
    PropertyEditorSupport --> PropertyEditor
```

```
public class PropertyEditorSupport implements PropertyEditor {  
    // Protected Constructors  
    protected PropertyEditorSupport();  
    protected PropertyEditorSupport(Object source);  
    // Event Registration Methods (by event name)  
    public void addPropertyChangeListener(PropertyChangeListener listener); Implements:PropertyEditor synchronized  
    public void removePropertyChangeListener(PropertyChangeListener listener); Implements:PropertyEditor synchronized
```

```
// Public Instance Methods
public void firePropertyChange();
// Methods Implementing PropertyEditor
public void addPropertyChangeListener(PropertyChangeListener listener);           synchronized
public String getAsText();
public java.awt.Component getCustomEditor();                                   constant
public String getJavaInitializationString();
public String[] getTags();                                                    constant
public Object getValue();
public boolean isPaintable();                                                  constant
public void paintValue(java.awt.Graphics gfx, java.awt.Rectangle box);          empty
public void removePropertyChangeListener(PropertyChangeListener listener);    synchronized
public void setAsText(String text) throws IllegalArgumentException;
public void setValue(Object value);
public boolean supportsCustomEditor();                                         constant
}
```

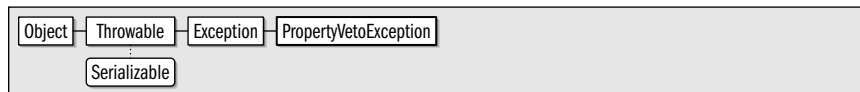
## PropertyVetoException

Java 1.1

java.beans

*serializable checked*

A `PropertyVetoException` signals that a `VetoableChangeListener` that received a `PropertyChangeEvent` for a constrained property of a bean has vetoed that proposed change. When this exception is received, the property in question should revert to its original value, and any `VetoableChangeListener` objects that have already been notified of the property change must be renotified to indicate that the property has reverted to its old value. The `VetoableChangeSupport` class handles this renotification automatically and rethrows the `PropertyVetoException` to notify its caller that the change was rejected.



```
public class PropertyVetoException extends Exception {
// Public Constructors
public PropertyVetoException(String mess, PropertyChangeEvent evt);
// Public Instance Methods
public PropertyChangeEvent getPropertyChangeEvent();
}
```

*Thrown By:* Too many methods to list.

## SimpleBeanInfo

Java 1.1

java.beans

The `SimpleBeanInfo` class is a trivial implementation of the `BeanInfo` interface. The methods of this class all return `null` or `-1`, indicating that no bean information is available. To use this class, you need to override only the method or methods that return the particular type of bean information you want to provide. In addition, `SimpleBeanInfo` provides a convenience method, `loadImage()`, that takes a resource name as an argument and returns an `Image` object. This method is useful when defining the `getIcon()` method.



```
public class SimpleBeanInfo implements BeanInfo {
// Public Constructors
public SimpleBeanInfo();
}
```

## SimpleBeanInfo

```
// Public Instance Methods
public java.awt.Image loadImage(String resourceName);
// Methods Implementing BeanInfo
public BeanInfo[] getAdditionalBeanInfo();           constant default:null
public BeanDescriptor getBeanDescriptor();          constant default:null
public int getDefaultEventIndex();                  constant default:-1
public int getDefaultPropertyIndex();               constant default:-1
public EventSetDescriptor[] getEventSetDescriptors(); constant default:null
public java.awt.Image getIcon(int iconKind);         constant
public MethodDescriptor[] getMethodDescriptors();  constant default:null
public PropertyDescriptor[] getPropertyDescriptors(); constant default:null
}
```

### Statement

Java 1.4

java.beans

This simple class represents the invocation of a method that has no return value. Within the framework of the JavaBeans persistence mechanism, **Statement** objects are generated by a **PersistenceDelegate** and translated to some textual form by an **Encoder**, such as the **XMLEncoder** class.

To create a **Statement** object, specify the target object upon which the method is to be called, the name of the method to invoke, and an array of arguments to pass to the method. If any of the method arguments are primitive values, use a corresponding wrapper object: use an **Integer** to represent an int value, for example. To invoke a static method, use the appropriate **Class** object as the target object. If the target object is an array, you can set an element of the array by using the method name “set”, passing an **Integer** to specify the array index and an **Object** to specify the value to be stored at that index. **execute()** uses the **java.lang.reflect** package to invoke the named method on the specified target with the specified arguments. See also the subclass **Expression** which describes a method invocation with a return value.

```
public class Statement {
// Public Constructors
    public Statement(Object target, String methodName, Object[] arguments);
// Public Instance Methods
    public void execute() throws Exception;
    public Object[] getArguments();
    public String getMethodName();
    public Object getTarget();
// Public Methods Overriding Object
    public String toString();
}
```

**Subclasses:** Expression

**Passed To:** Encoder.writeStatement(), XMLEncoder.writeStatement()

### VetoableChangeListener


Java 1.1

java.beans

event listener

This interface is an extension of **java.util.EventListener**. It defines the method a class must implement in order to be notified when a Java bean makes a change to a constrained property. A **PropertyChangeEvent** is passed to the **vetoableChange()** method when such a

change occurs. If the `VetoableChangeListener` wants to prevent the change from occurring, this method should throw a `PropertyVetoException`.



```

public interface VetoableChangeListener extends java.util.EventListener {
    // Public Instance Methods
    public abstract void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException;
}

```

**Implementations:** `VetoableChangeListenerProxy`, `java.beans.beancontext.BeanContextSupport`

**Passed To:** `java.awt.KeyboardFocusManager.addVetoableChangeListener()`, `removeVetoableChangeListener()`, `VetoableChangeListenerProxy.VetoableChangeListenerProxy()`, `VetoableChangeSupport.addVetoableChangeListener()`, `removeVetoableChangeListener()`, `java.beans.beancontext.BeanContextChild.addVetoableChangeListener()`, `removeVetoableChangeListener()`, `java.beans.beancontext.BeanContextChildSupport.addVetoableChangeListener()`, `removeVetoableChangeListener()`, `javax.swing.JComponent.addVetoableChangeListener()`, `removeVetoableChangeListener()`

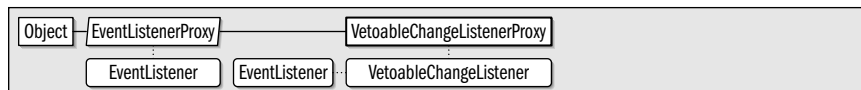
**Returned By:** `java.awt.KeyboardFocusManager.getVetoableChangeListeners()`, `VetoableChangeSupport.getVetoableChangeListeners()`, `java.beans.beancontext.BeanContextSupport.getChildVetoableChangeListener()`, `javax.swing.JComponent.getVetoableChangeListeners()`

## VetoableChangeListenerProxy

Java 1.4

`java.beans`

This class implements `VetoableChangeListener` and serves as a wrapper around another `VetoableChangeListener` object. Beans that define a two-argument `addVetoableChangeListener()` method to allow registration of a listener for a specific named property may return instances of this class from their `getVetoableChangeListeners()` method (if they implement one). Use `getPropertyName()` to determine the name of the property to which the listener applies. And use the inherited `getListener()` method to obtain the underlying `VetoableChangeListener` object.



```

public class VetoableChangeListenerProxy extends java.util.EventListenerProxy
    implements VetoableChangeListener {
    // Public Constructors
    public VetoableChangeListenerProxy(String propertyName, VetoableChangeListener listener);
    // Public Instance Methods
    public String getPropertyName();
    // Methods Implementing VetoableChangeListener
    public void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException;
}

```

## VetoableChangeSupport

Java 1.1

`java.beans`

*serializable*

`VetoableChangeSupport` is a convenience class that maintains a list of registered `VetoableChangeListener` objects and provides a `fireVetoableChange()` method for sending a `PropertyChangeEvent` to all registered listeners. If any of the registered listeners veto the proposed change, `fireVetoableChange()` sends out another `PropertyChangeEvent` notifying

## VetoableChangeSupport

previously notified listeners that the property has reverted to its original value. Because of the extra complexity of correctly handling veto-able changes and because of some tricky thread-synchronization issues involved in maintaining the list of listeners, it is recommended that all Java beans that support constrained events create a `VetoableChangeSupport` object to which they can delegate their `addVetoableChangeListener()` and `removeVetoableChangeListener()` methods. In Java 1.4, beans can also define a `getVetoableChangeListeners()` method and delegate it to a `VetoableChangeSupport` object.

```
Object VetoableChangeSupport Serializable

public class VetoableChangeSupport implements Serializable {
    // Public Constructors
    public VetoableChangeSupport(Object sourceBean);
    // Event Registration Methods (by event name)
    public void addVetoableChangeListener(VetoableChangeListener listener);           synchronized
    public void removeVetoableChangeListener(VetoableChangeListener listener);       synchronized
    // Public Instance Methods
    1.2 public void addVetoableChangeListener(String propertyName, VetoableChangeListener listener); synchronized
    1.2 public void fireVetoableChange(PropertyChangeEvent evt) throws PropertyVetoException;
    1.2 public void fireVetoableChange(String propertyName, int oldValue, int newValue) throws PropertyVetoException;
    public void fireVetoableChange(String propertyName, Object oldValue, Object newValue)
        throws PropertyVetoException;
    1.2 public void fireVetoableChange(String propertyName, boolean oldValue, boolean newValue)
        throws PropertyVetoException;
    1.4 public VetoableChangeListener[] getVetoableChangeListeners();                 synchronized
    1.4 public VetoableChangeListener[] getVetoableChangeListeners(String propertyName); synchronized
    1.2 public boolean hasListeners(String propertyName);                           synchronized
    1.2 public void removeVetoableChangeListener(String propertyName,
        VetoableChangeListener listener);                                           synchronized
}
```

Type Of: `java.beans.beancontext.BeanContextChildSupport.vcSupport`

## Visibility

Java 1.1

### java.beans

This interface is intended to be implemented by advanced beans that can run both with and without a GUI present. The methods it defines allow a bean to specify whether it requires a GUI and allow the environment to notify the bean whether a GUI is available. If a bean absolutely requires a GUI, it should return `true` from `needsGui()`. If a bean is running without a GUI, it should return `true` from `avoidingGui()`. If no GUI is available, the bean can be notified through a call to `dontUseGui()`, and if a GUI is available, the bean can be notified through a call to `okToUseGui()`.

```
public interface Visibility {
    // Public Instance Methods
    public abstract boolean avoidingGui();
    public abstract void dontUseGui();
    public abstract boolean needsGui();
    public abstract void okToUseGui();
}
```

Implementations: `java.beans.beancontext.BeanContext`

Returned By: `java.beans.beancontext.BeanContextSupport.getChildVisibility()`

**XMLDecoder**

Java 1.4

**java.beans**

This class recreates JavaBeans that were stored in XML format by an `XMLEncoder`. Create an `XMLDecoder` by specifying the `java.io.InputStream` that the XML-encoded serialized beans are to be read from. Then call `readObject()` one or more times to read encoded beans from that file. `readObject()` throws an `ArrayIndexOutOfBoundsException` when there are no more beans to read. Call `close()` to close the underlying stream when you are done with an `XMLDecoder`. Because the `XMLEncoder` encodes the state of a bean as a series of public method calls, the decoding process is relatively efficient, and there is no need for the decoding application to load, or even have access to, the various `PersistenceDelegate` classes that were used to encode the bean.

The bean decoding process is designed to be robust and tries to recover from any exceptions that are thrown whenever possible. Pass an `ExceptionListener` to the `XMLDecoder()` constructor if you want to receive notification of exceptions that occur during the decoding process. (An `XMLDecoder` begins decoding as soon as the constructor is called, so it is not useful to pass an `ExceptionListener` to `setExceptionListener()`: any such call is too late.)

If the stream you are decoding included a call to `setOwner()` when it was encoded, then you must pass an owner object to the `XMLDecoder()` constructor so that encoded method calls that use the owner object can be decoded. (There is a `setOwner()` method, but like `setExceptionListener()`, it is not useful.)

```
public class XMLDecoder {
// Public Constructors
    public XMLDecoder(java.io.InputStream in);
    public XMLDecoder(java.io.InputStream in, Object owner);
    public XMLDecoder(java.io.InputStream in, Object owner, ExceptionListener exceptionListener);
// Public Instance Methods
    public void close();
    public ExceptionListener getExceptionListener();
    public Object getOwner();
    public Object readObject();
    public void setExceptionListener(ExceptionListener exceptionListener);
    public void setOwner(Object owner);
}
```

**XMLEncoder**

Java 1.4

**java.beans**

This class creates an XML-formatted description of a `JavaBean` or of a tree of beans and writes them to a stream. Specify the stream to be written to when you create the `XMLEncoder`. Then call the `writeObject()` method one or more times to serialize your `JavaBeans` using the XML persistence format. Note that `writeObject()` serializes the specified object, and, recursively, any objects it refers to in its public API. It is important to call `close()` when finished so that the `XMLEncoder` can flush its internal buffers, output a closing XML tag, and flush and close the stream. You can also `flush()` the encoder without closing it, if necessary. Use `XMLDecoder` to recreate any beans encoded with this class. Contrast this class with the `java.io.ObjectOutputStream` which performs serialization using a binary format.

The bean encoding process is designed to be robust and tries to recover from internal exceptions whenever possible. Use the inherited `setExceptionListener()` method if you want to register an `ExceptionListener` to receive notification of exceptions that occur and are handled during the encoding process.



## XMLEncoder

The bean encoding process uses a `PersistenceDelegate` object internally. The `DefaultPersistenceDelegate` class is used for all conforming JavaBeans that have a no-argument constructor and whose entire state is encapsulated by properties that have public get and set accessor methods. The implementation also provides private persistence delegates for all `java.awt.Component` classes from the AWT and Swing packages (covered in the companion book *Java Foundation Classes in a Nutshell*), as well as all classes (such as `java.awt.Font` and `java.awt.Color`) that those components use as properties. If you want to encode a custom bean that does not strictly follow the JavaBeans constructor and property conventions, you need to create a custom `PersistenceDelegate` object and associate it with the `Class` object for the custom bean by calling the inherited `setPersistenceDelegate()` method. See `PersistenceDelegate` for details.

In addition to encoding the state of a bean, this class can also be used to encode arbitrary method calls with `writeExpression()` and `writeStatement()`. This is particularly useful when used with the `setOwner()` method. Use `setOwner()` to specify an object that is not to be serialized as part of the bean state. Then, encode method calls on this owner object with `writeStatement()`. When the XML archive is decoded with an `XMLDecoder`, you'll use a similar `setOwner()` method to specify the object on which those methods should be invoked. When the owner object represents the application backend, this is a powerful way to tie a serialized GUI to that backend by registering event listeners, or passing particular beans to the backend for manipulation.

Object	Encoder	XMLEncoder
--------	---------	------------

```
public class XMLEncoder extends Encoder {  
    // Public Constructors  
    public XMLEncoder(java.io.OutputStream out);  
    // Public Instance Methods  
    public void close();  
    public void flush();  
    public Object getOwner();  
    public void setOwner(Object owner);  
    // Public Methods Overriding Encoder  
    public void writeExpression(Expression oldExp);  
    public void writeObject(Object o);  
    public void writeStatement(java.beans.Statement oldStm);  
}
```

## Package java.beans.beancontext

Java 1.2

The `java.beans.beancontext` package extends the JavaBeans component model to add the notion of a containment hierarchy. It also supports bean containers that provide an execution context for the beans they contain and that may also provide a set of services to those beans. This package is typically used by advanced bean developers and developers of bean-manipulation tools. Application programmers who are simply using beans do not typically use this package.

`BeanContext` is the central interface of this package. It is a container for beans and also defines several methods that specify context information for beans. `BeanContextServices` extends `BeanContext` to define methods that allow a contained bean to query and request available services. A bean that wishes to be told about its containing `BeanContext` implements the `BeanContextChild` interface. `BeanContext` is itself a `BeanContextChild`, which means that contexts can be nested within other contexts.

See Chapter 6 for more information on beans and bean contexts.

*Interfaces:*

```
public interface BeanContextChild;
public interface BeanContextChildComponentProxy;
public interface BeanContextContainerProxy;
public interface BeanContextProxy;
public interface BeanContextServiceProvider;
public interface BeanContextServiceProviderBeanInfo extends java.beans.BeanInfo;
```

*Collections:*

```
public interface BeanContext extends BeanContextChild, java.util.Collection,
    java.beans.DesignMode, java.beans.Visibility;
public class BeanContextSupport extends BeanContextChildSupport
    implements BeanContext, java.beans.PropertyChangeListener,
    Serializable, java.beans.VetoableChangeListener;
    public class BeanContextServicesSupport extends BeanContextSupport
        implements BeanContextServices;
```

*Events:*

```
public abstract class BeanContextEvent extends java.util.EventObject;
    public class BeanContextMembershipEvent extends BeanContextEvent;
    public class BeanContextServiceAvailableEvent extends BeanContextEvent;
    public class BeanContextServiceRevokedEvent extends BeanContextEvent;
```

*Event Listeners:*

```
public interface BeanContextMembershipListener extends java.util.EventListener;
public interface BeanContextServiceRevokedListener extends java.util.EventListener;
public interface BeanContextServices extends BeanContext, BeanContextServicesListener;
public interface BeanContextServicesListener extends BeanContextServiceRevokedListener;
```

*Other Classes:*

```
public class BeanContextChildSupport
    implements BeanContextChild, BeanContextServicesListener, Serializable;
```

*Protected Inner Classes:*

```
protected class BeanContextServicesSupport.BCSSProxyServiceProvider
    implements BeanContextServiceProvider, BeanContextServiceRevokedListener;
protected static class BeanContextServicesSupport.BCSSServiceProvider implements Serializable;
protected class BeanContextSupport.BCSChild implements Serializable;
    protected class BeanContextServicesSupport.BCSSChild extends BeanContextSupport.BCSChild;
    protected static final class BeanContextSupport.BCSIterator implements java.util.Iterator;
```

## BeanContext

Java 1.2

java.beans.beancontext


collection

This interface defines the methods that must be implemented by any class that wants to act as a logical container for JavaBeans components. Every **BeanContext** is also a **BeanContextChild** and can therefore be nested within a higher-level bean context. **BeanContext** is extended by **BeanContextServices**; any bean context that wants to provide services to the beans it contains must implement this more specialized interface.

## BeanContext

The **BeanContext** interface extends the `java.util.Collection` interface; the children it contains are accessed using the methods of that interface. In addition, **BeanContext** defines several important methods of its own. `instantiateChild()` instantiates a new bean, in the same manner as the standard `Beans.instantiate()` method, and then makes that new bean a child of the context. Calling this method is typically the same as calling the three-argument version of `Beans.instantiate()`. `getResource()` and `getResourceAsStream()` are the **BeanContext** versions of the `java.lang.Class` and `java.lang.ClassLoader` methods of the same name. Some bean-context implementations may provide special behavior for these methods; others may simply delegate to the `Class` or `ClassLoader` of the bean. The remaining two methods allow the registration and deregistration of event listeners that the **BeanContext** notifies when bean children are added or removed from the context.

Implementing a **BeanContext** is a more specialized task than developing a JavaBeans component. Many bean developers will never have to implement a bean context themselves. If you do implement a bean context, you'll probably find it easier to use **BeanContextSupport**, either by extending it or using an instance as a proxy.



```
public interface BeanContext extends BeanContextChild, java.util.Collection, java.beans.DesignMode,
    java.beans.Visibility {
    // Public Constants
    public static final Object globalHierarchyLock;
    // Event Registration Methods (by event name)
    public abstract void addBeanContextMembershipListener(BeanContextMembershipListener bcmf);
    public abstract void removeBeanContextMembershipListener(BeanContextMembershipListener bcmf);
    // Public Instance Methods
    public abstract java.net.URL getResource(String name, BeanContextChild bcc) throws IllegalArgumentException;
    public abstract java.io.InputStream getResourceAsStream(String name, BeanContextChild bcc)
        throws IllegalArgumentException;
    public abstract Object instantiateChild(String beanName) throws java.io.IOException, ClassNotFoundException;
}
```

**Implementations:** **BeanContextServices**, **BeanContextSupport**

**Passed To:** `java.beans.AppletInitializer.initialize()`, `java.beans.Beans.instantiate()`, `BeanContextChild.setBeanContext()`, `BeanContextChildSupport.setBeanContext()`, `validatePendingSetBeanContext()`, `BeanContextEvent.getBeanContext()`, `setPropagatedFrom()`, `BeanContextMembershipEvent.getBeanContext()`, `BeanContextMembershipEvent.getBeanContext()`, `BeanContextSupport.getBeanContext()`

**Returned By:** `BeanContextChild.getBeanContext()`, `BeanContextChildSupport.getBeanContext()`, `BeanContextEvent.getBeanContext()`, `getPropagatedFrom()`, `BeanContextSupport.getBeanContextPeer()`

**Type Of:** `BeanContextChildSupport.getBeanContext`, `BeanContextEvent.propagatedFrom`

## BeanContextChild

Java 1.2

`java.beans.beancontext`

JavaBeans components that are designed to be nested within a bean context and need to be aware of that context must implement this interface. **BeanContextChild** implements a single `beanContext` property that identifies the **BeanContext** within which the bean is contained. The `beanContext` property is bound and constrained, which means that it must fire `PropertyChangeEvent` events when `setBeanContext()` is called, and any call to `setBeanContext()` may result in a `PropertyVetoException` if one of the `VetoableChangeListener` objects vetoes the change. The `setBeanContext()` method is not intended for use by beans or by

applications. When a bean is instantiated or deserialized, its containing bean context calls this method to introduce itself to the bean. The bean must store a reference to its `BeanContext` in a `transient` field so that the context is not serialized along with the bean itself.

Implementing a `BeanContextChild` from scratch can be somewhat tricky because you must correctly handle the `VetoableChangeListener` protocol and correctly implement important conventions, such as storing the `BeanContext` reference in a `transient` field. Therefore, most bean developers do not implement the interface directly, but instead use `BeanContextSupport`, either by subclassing it or by using an instance as a delegate.

```
public interface BeanContextChild {
    // Public Instance Methods
    public abstract void addPropertyChangeListener(String name, java.beans.PropertyChangeListener pcl);
    public abstract void addVetoableChangeListener(String name, java.beans.VetoableChangeListener vc);
    public abstract BeanContext getBeanContext();
    public abstract void removePropertyChangeListener(String name, java.beans.PropertyChangeListener pcl);
    public abstract void removeVetoableChangeListener(String name, java.beans.VetoableChangeListener vc);
    public abstract void setBeanContext(BeanContext bc) throws java.beans.PropertyVetoException;
}
```

*Implementations:* `BeanContext`, `BeanContextChildSupport`

*Passed To:* `BeanContext`.{`getResource()`, `getResourceAsStream()`},  
`BeanContextChildSupport`.`BeanContextChildSupport()`, `BeanContextServices`.{`getService()`,  
`releaseService()`}, `BeanContextServicesSupport`.{`getService()`, `releaseService()`},  
`BeanContextSupport`.{`getResource()`, `getResourceAsStream()`}

*Returned By:* `BeanContextChildSupport`.`getBeanContextChildPeer()`,  
`BeanContextProxy`.`getBeanContextProxy()`, `BeanContextSupport`.`getChildBeanContextChild()`

*Type Of:* `BeanContextChildSupport`.`beanContextChildPeer`

## **BeanContextChildComponentProxy**

**Java 1.2**

`java.beans.beancontext`

If a `BeanContextChild` is not a `Component` subclass but has an associated `Component` object to display its visual representation, it implements this interface to allow access to that component.

```
public interface BeanContextChildComponentProxy {
    // Public Instance Methods
    public abstract java.awt.Component getComponent();
}
```

## **BeanContextChildSupport**

**Java 1.2**

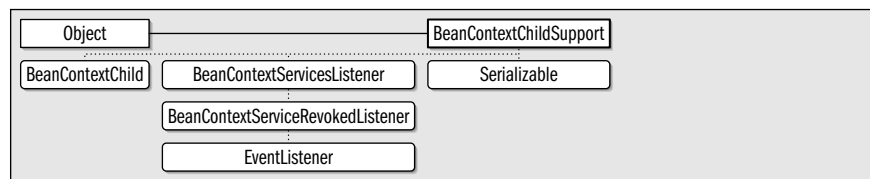
`java.beans.beancontext`

*serializable*

This class provides support for implementing the `BeanContextChild` interface in a way that correctly conforms to the details of the bean context specification. A subclass should implement `initializeBeanContextResources()` and `releaseBeanContextResources()` to obtain and release whatever resources the bean context child requires, such as service objects obtained from the containing `BeanContext`. These methods are called when the containing bean context introduces itself by calling `setBeanContext()`. Any resources obtained with these methods should be stored in `transient` fields so that they are not serialized along with the bean. A bean that wants a chance to approve any call to `setBeanContext()` before that call succeeds can implement `validatePendingSetBeanContext()`. If this method returns `false`, the `setBeanContext()` call that triggered it fails with a `PropertyVetoException`.

## BeanContextChildSupport

Many beans are AWT or Swing components and cannot subclass both `Component` and `BeanContextChildSupport`. Therefore, many bean developers find it useful to delegate to an internal instance of `BeanContextChildSupport`. One way to do this is to have your bean implement the `BeanContextProxy` interface and simply return an instance of `BeanContextChildSupport` from the `getBeanContextProxy()` method. Another technique is to actually implement the `BeanContextChild` interface in your bean, but provide dummy methods that call the corresponding methods of `BeanContextChildSupport`. If you do this, you should pass an instance of your bean to the `BeanContextChildSupport()` constructor. This makes the delegation transparent so events appear to come directly from your bean. In either case, you can instantiate `BeanContextChildSupport` directly. Often, however, you want to create a custom subclass (perhaps as an inner class) to implement methods such as `initializeBeanContextResources()`.



```

public class BeanContextChildSupport implements BeanContextChild, BeanContextServicesListener, Serializable {
    // Public Constructors
    public BeanContextChildSupport();
    public BeanContextChildSupport(BeanContextChild bcc);
    // Public Instance Methods
    public void firePropertyChange(String name, Object oldValue, Object newValue);
    public void fireVetoableChange(String name, Object oldValue, Object newValue)
        throws java.beans.PropertyVetoException;
    public BeanContextChild getBeanContextChildPeer(); default:BeanContextChildSupport
    public boolean isDelegated(); default:false
    public boolean validatePendingSetBeanContext(BeanContext newValue); constant
    // Methods Implementing BeanContextChild
    public void addPropertyChangeListener(String name, java.beans.PropertyChangeListener pcl);
    public void addVetoableChangeListener(String name, java.beans.VetoableChangeListener vc);
    public BeanContext getBeanContext(); synchronized default:null
    public void removePropertyChangeListener(String name, java.beans.PropertyChangeListener pcl);
    public void removeVetoableChangeListener(String name, java.beans.VetoableChangeListener vc);
    public void setBeanContext(BeanContext bc) throws java.beans.PropertyVetoException; synchronized
    // Methods Implementing BeanContextServiceRevokedListener
    public void serviceRevoked(BeanContextServiceRevokedEvent bcsre); empty
    // Methods Implementing BeanContextServicesListener
    public void serviceAvailable(BeanContextServiceAvailableEvent bcsae); empty
    // Protected Instance Methods
    protected void initializeBeanContextResources(); empty
    protected void releaseBeanContextResources(); empty
    // Public Instance Fields
    public BeanContextChild beanContextChildPeer;
    // Protected Instance Fields
    protected transient BeanContext beanContext;
    protected java.beans.PropertyChangeSupport pcSupport;
    protected transient boolean rejectedSetBCOnce;
    protected java.beans.VetoableChangeSupport vcSupport;
}

```

*Subclasses:* `BeanContextSupport`

## BeanContextContainerProxy

Java 1.2

java.beans.beancontext

This interface is implemented by a `BeanContext` that has a `java.awt.Container` associated with it. The `getContainer()` method allows any interested parties to obtain a reference to the container associated with the bean context. It is a common practice for bean contexts to be associated with containers. Unfortunately, `BeanContext` implements `java.util.Collection`, which has method-name conflicts with `java.awt.Container`, so no `Container` subclass can implement the `BeanContext` interface. See also `BeanContextProxy`, which reverses the direction of the proxy relationship.

```
public interface BeanContextContainerProxy {
    // Public Instance Methods
    public abstract java.awt.Container getContainer();
}
```

## BeanContextEvent

Java 1.2

java.beans.beancontext

serializable event

This is the abstract superclass of all bean context-related events. `getBeanContext()` returns the source of the event. If `isPropagated()` returns `true`, the event has been propagated through a hierarchy of bean contexts, and `getPropagatedFrom()` returns the most recent bean context to propagate the event.



```
public abstract class BeanContextEvent extends java.util.EventObject {
    // Protected Constructors
    protected BeanContextEvent(BeaContext bc);
    // Public Instance Methods
    public BeaContext getBeaContext();
    public BeaContext getPropagatedFrom();
    public boolean isPropagated();
    public void setPropagatedFrom(BeaContext bc);
    // Protected Instance Fields
    protected BeaContext propagatedFrom;
}
```

*Subclasses:* `BeanContextMembershipEvent`, `BeanContextServiceAvailableEvent`, `BeanContextServiceRevokedEvent`

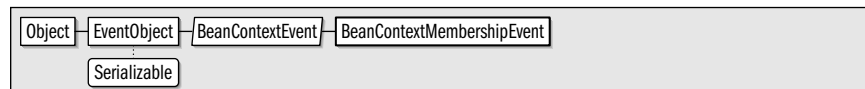
## BeanContextMembershipEvent

Java 1.2

java.beans.beancontext

serializable event

An event of this type is generated by a `BeanContext` when beans are added to it or removed from it. The event object contains the list of children that were added or removed and allows access to that list in several ways. The `size()` method returns the number of affected children. The `contains()` method checks whether a specified object was one of the affected children. `toArray()` returns the list of affected children as an array, and `iterator()` returns the list in the form of a `java.util.Iterator`.



## *BeanContextMembershipEvent*

```
public class BeanContextMembershipEvent extends BeanContextEvent {  
    // Public Constructors  
    public BeanContextMembershipEvent(BeanContext bc, Object[ ] changes);  
    public BeanContextMembershipEvent(BeanContext bc, java.util.Collection changes);  
    // Public Instance Methods  
    public boolean contains(Object child);  
    public java.util.Iterator iterator();  
    public int size();  
    public Object[ ] toArray();  
    // Protected Instance Fields  
    protected java.util.Collection children;  
}
```

*Passed To:* BeanContextMembershipListener.{childrenAdded(), childrenRemoved()},  
BeanContextSupport.{fireChildrenAdded(), fireChildrenRemoved()}

## **BeanContextMembershipListener**

**Java 1.2**

java.beans.beancontext

*event listener*

This interface should be implemented by any object that wants to be notified when children are added to or removed from a BeanContext.

EventListener · BeanContextMembershipListener

```
public interface BeanContextMembershipListener extends java.util.EventListener {  
    // Public Instance Methods  
    public abstract void childrenAdded(BeanContextMembershipEvent bcme);  
    public abstract void childrenRemoved(BeanContextMembershipEvent bcme);  
}
```

*Passed To:* BeanContext.{addBeanContextMembershipListener(),  
removeBeanContextMembershipListener()},  
BeanContextSupport.{addBeanContextMembershipListener(),  
removeBeanContextMembershipListener()}

*Returned By:* BeanContextSupport.getChildBeanContextMembershipListener()

## **BeanContextProxy**

**Java 1.2**

java.beans.beancontext

This interface is implemented by a JavaBeans component (often, but not always, an AWT Component or Container object) that is not itself a BeanContext or BeanContextChild, but has a BeanContext or BeanContextChild object associated with it. The `getBeanContextProxy()` method returns the associated object. The return type of this method is BeanContextChild. Depending on the context in which you call this method, however, the returned object may actually be a BeanContext or BeanContextServices object. You should test for this using the instanceof operator before casting the object to these more specific types.

```
public interface BeanContextProxy {  
    // Public Instance Methods  
    public abstract BeanContextChild getBeanContextProxy();  
}
```

## BeanContextServiceAvailableEvent

Java 1.2

java.beans.beancontext

serializable event

An event of this type is generated to notify interested `BeanContextServicesListener` objects that a new class of service is available from a `BeanContextServices` object. `getServiceClass()` returns the class of the service, and `getCurrentServiceSelectors()` may return a set of additional arguments that can parameterize the service.



```

public class BeanContextServiceAvailableEvent extends BeanContextEvent {
// Public Constructors
    public BeanContextServiceAvailableEvent(BeanContextServices bcs, Class sc);
// Public Instance Methods
    public java.util.Iterator getCurrentServiceSelectors();
    public Class getServiceClass();
    public BeanContextServices getSourceAsBeanContextServices();
// Protected Instance Fields
    protected Class serviceClass;
}
  
```

*Passed To:* `BeanContextChildSupport.serviceAvailable()`,  
`BeanContextServicesListener.serviceAvailable()`, `BeanContextServicesSupport.fireServiceAdded()`,  
`serviceAvailable()`

## BeanContextServiceProvider

Java 1.2

java.beans.beancontext

This interface defines the methods that must be implemented by a factory class that wants to provide service objects to beans. To provide its service, a `BeanContextServiceProvider` is passed to the `addService()` method of a `BeanContextServices` object. This creates a mapping in the `BeanContextServices` object between a class of service (such as `java.awt.print.PrinterJob`) and a `BeanContextServiceProvider` that can return a suitable instance of that class to provide the service.

When a `BeanContextChild` requests a service of a particular class from its `BeanContextServices` container, the `BeanContextServices` object finds the appropriate `BeanContextServiceProvider` object and forwards the request to its `getService()` method. When the bean relinquishes the service, `releaseService()` is called. A `getService()` request may include an arbitrary object as an additional parameter or service selector. Service providers that use the service selector argument and that support a finite set of legal service selector values should implement the `getCurrentServiceSelectors()` method to allow the list of legal selector values to be queried.

Bean developers typically do not have to use or implement this interface. From the point of view of a bean context child, service objects are obtained from a `BeanContextServices` object. Developers creating `BeanContextServices` implementations, however, must implement appropriate `BeanContextServiceProvider` objects to provide the services.

```

public interface BeanContextServiceProvider {
// Public Instance Methods
    public abstract java.util.Iterator getCurrentServiceSelectors(BeanContextServices bcs, Class serviceClass);
    public abstract Object getService(BeanContextServices bcs, Object requestor, Class serviceClass,
                                     Object serviceSelector);
    public abstract void releaseService(BeanContextServices bcs, Object requestor, Object service);
}
  
```



## BeanContextServiceProvider

**Implementations:** BeanContextServicesSupport.BCSSProxyServiceProvider

**Passed To:** BeanContextServices.{addService(), revokeService()},  
BeanContextServicesSupport.{addService(), createBCSSServiceProvider(), revokeService()}

**Returned By:** BeanContextServicesSupport.BCSSServiceProvider.getServiceProvider()

**Type Of:** BeanContextServicesSupport.BCSSServiceProvider.serviceProvider

## BeanContextServiceProviderBeanInfo

Java 1.2

java.beans.beancontext

A **BeanContextServiceProvider** that wishes to provide information to a GUI builder tool about the service or services it offers should implement this **BeanInfo** subinterface. Following the standard **BeanInfo** naming conventions, the implementing class should have the same name as the service provider class, with “**BeanInfo**” appended. This enables a design tool to look for and dynamically load the bean info class when necessary.

**getServicesBeanInfo()** should return an array of **BeanInfo** objects, one for each class of service offered by the **BeanContextServiceProvider**. These **BeanInfo** objects enable a design tool to allow the user to visually configure the service object. This can be quite useful, since service objects may be instances of existing classes that were not designed with the standard JavaBeans naming conventions in mind.

BeanInfo BeanContextServiceProviderBeanInfo

```
public interface BeanContextServiceProviderBeanInfo extends java.beans.BeanInfo {  
    // Public Instance Methods  
    public abstract java.beans.BeanInfo[] getServicesBeanInfo();  
}
```

## BeanContextServiceRevokedEvent

Java 1.2

java.beans.beancontext

serializable event

This event class provides details about a service revocation initiated by a **BeanContextServices** object. **getServiceClass()** specifies the class of service being revoked. **isCurrentServiceInvalidNow()** specifies whether the currently owned service object has become invalid. If this method returns **true**, the bean that receives this event must stop using the service object immediately. If the method returns **false**, the bean can continue to use the service object, but future requests for services of this class will fail.

Object EventObject BeanContextEvent BeanContextServiceRevokedEvent  
Serializable

```
public class BeanContextServiceRevokedEvent extends BeanContextEvent {  
    // Public Constructors  
    public BeanContextServiceRevokedEvent(BeanContextServices bcs, Class sc, boolean invalidate);  
    // Public Instance Methods  
    public Class getServiceClass();  
    public BeanContextServices getSourceAsBeanContextServices();  
    public boolean isCurrentServiceInvalidNow();  
    public boolean isServiceClass(Class service);  
    // Protected Instance Fields  
    protected Class serviceClass;  
}
```

**Passed To:** BeanContextChildSupport.serviceRevoked(),  
BeanContextServiceRevokedListener.serviceRevoked(),

```
BeanContextServicesSupport.fireServiceRevoked(), serviceRevoked()),
BeanContextServicesSupport.BCSSProxyServiceProvider.serviceRevoked()
```

## BeanContextServiceRevokedListener

Java 1.2

java.beans.beancontext

event listener

This interface defines a method that is invoked when a service object returned by a `BeanContextServices` object is forcibly revoked. Unlike other types of event listeners, the `BeanContextServiceRevokedListener` is not registered and deregistered with a pair of add and remove methods. Instead, an implementation of this interface must be passed to every `getService()` call on a `BeanContextServices` object. If the returned service is ever revoked by the granting `BeanContextServiceProvider` object before the bean has relinquished the service, the `serviceRevoked()` method of this interface is called.

When a service is revoked, it means that future requests for the service will not succeed. But it may also mean that current service objects have become invalid and must not be used anymore. The `serviceRevoked()` method should call the `isCurrentServiceInvalidNow()` method of the supplied event object to determine if this is the case. If so, it must immediately stop using the service object.

EventListener BeanContextServiceRevokedListener

```
public interface BeanContextServiceRevokedListener extends java.util.EventListener {
    // Public Instance Methods
    public abstract void serviceRevoked(BeansContextServiceRevokedEvent bcsre);
}
```

**Implementations:** `BeanContextServicesListener`,  
`BeanContextServicesSupport.BCSSProxyServiceProvider`

**Passed To:** `BeanContextServices.getService()`, `BeanContextServicesSupport.getService()`

## BeanContextServices

Java 1.2

java.beans.beancontext

collection event listener

This interface defines additional methods a bean context class must implement if it wants to provide services to the beans it contains. A bean calls `hasService()` to determine if a service of a particular type is available from its bean context. It calls `getService()` to request an instance of the specified service class and then calls `releaseService()` when it no longer needs the service object. A bean that wants to find the complete list of available services can call `getCurrentServiceClasses()`. Some services allow (or require) a service selector object to be passed to the `getService()` method to provide additional information about the service object. If a service defines a fixed set of legal service selectors, `getCurrentServiceSelectors()` allows a bean to iterate through the set of selector objects. Beans that want to know when new services become available or when existing services are revoked should register a `BeanContextServicesListener` object with `addBeanContextServicesListener()`.

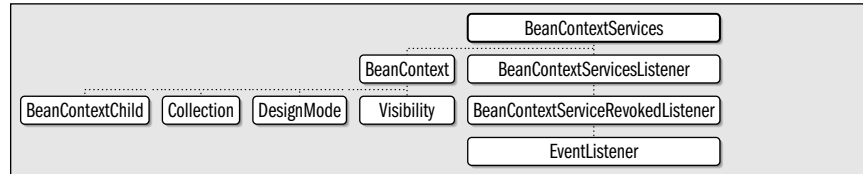
If a `BeanContextServices` object does not provide a requested service, but is nested within another `BeanContext`, it should check whether any of its ancestor bean contexts can provide the service. The `BeanContextServices` interface extends `BeanContextServicesListener`. This means that every `BeanContextServices` object can be listening to the set of services available from its container.

The previous methods are the ones beans call to obtain services. `BeanContextServices` defines a different set of methods service providers use to deliver services. `addService()` defines a `BeanContextServiceProvider` for a specified `Class` of service. `revokeService()` removes this mapping between service class and service provider, and indicates that the

## BeanContextServices

specified service is no longer available. When a service is revoked, the `BeanContextServices` object must notify any beans that have been granted service objects (and have not released them yet) that the service has been revoked. It does this by notifying the `BeanContextServiceRevokedListener` objects passed to the `getService()` method.

Bean context developers may find it easier to use the `BeanContextServicesSupport` class, either by subclassing or by delegation, instead of implementing `BeanContextServices` from scratch.



```

public interface BeanContextServices extends BeanContext, BeanContextServicesListener {
    // Event Registration Methods (by event name)
    public abstract void addBeanContextServicesListener(BeanContextServicesListener bcsf);
    public abstract void removeBeanContextServicesListener(BeanContextServicesListener bcsf);
    // Public Instance Methods
    public abstract boolean addService(Class serviceClass, BeanContextServiceProvider serviceProvider);
    public abstract java.util.Iterator getCurrentServiceClasses();
    public abstract java.util.Iterator getCurrentServiceSelectors(Class serviceClass);
    public abstract Object getService(BeanContextChild child, Object requestor, Class serviceClass,
        Object serviceSelector, BeanContextServiceRevokedListener bcsrf)
        throws java.util.TooManyListenersException;
    public abstract boolean hasService(Class serviceClass);
    public abstract void releaseService(BeanContextChild child, Object requestor, Object service);
    public abstract void revokeService(Class serviceClass, BeanContextServiceProvider serviceProvider,
        boolean revokeCurrentServicesNow);
}
  
```

**Implementations:** `BeanContextServicesSupport`

**Passed To:** `BeanContextServiceAvailableEvent.getBeanContextServiceAvailableEvent()`,  
`BeanContextServiceProvider.getCurrentServiceSelectors()`, `getService()`, `releaseService()`,  
`BeanContextServiceRevokedEvent.getBeanContextServiceRevokedEvent()`,  
`BeanContextServicesSupport.getBeanContextServicesSupport()`,  
`BeanContextServicesSupport.BCSPProxyServiceProvider.getCurrentServiceSelectors()`, `getService()`,  
`releaseService()`

**Returned By:** `BeanContextServiceAvailableEvent.getSourceAsBeanContextServices()`,  
`BeanContextServiceRevokedEvent.getSourceAsBeanContextServices()`,  
`BeanContextServicesSupport.getBeanContextServicesPeer()`

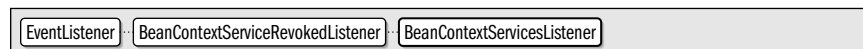
## BeanContextServicesListener

Java 1.2

`java.beans.beancontext`

event listener

This interface adds a `serviceAvailable()` method to the `serviceRevoked()` method of `BeanContextServiceRevokedListener`. Listeners of this type can be registered with a `BeanContextServices` object and are notified when a new class of service becomes available or when an existing class of service is revoked.



```

public interface BeanContextServicesListener extends BeanContextServiceRevokedListener {
    // Public Instance Methods
  
```

```
public abstract void serviceAvailable(BeanContextServiceAvailableEvent bcsae);
}
```

**Implementations:** BeanContextChildSupport, BeanContextServices

**Passed To:** BeanContextServices.{addBeanContextServicesListener(),  
removeBeanContextServicesListener()},  
BeanContextServicesSupport.{addBeanContextServicesListener(),  
removeBeanContextServicesListener()}

**Returned By:** BeanContextServicesSupport.getChildBeanContextServicesListener()

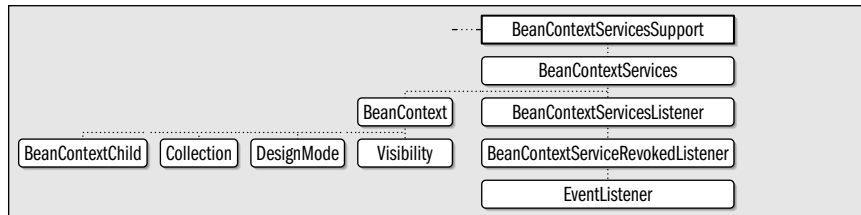
## BeanContextServicesSupport

Java 1.2

java.beans.beancontext

serializable collection

This class is a useful implementation of the `BeanContextServices` interface that correctly conforms to the bean context specifications and conventions. Most bean context implementors find it easier to subclass this class or delegate to an instance of this class rather than implement `BeanContextServices` from scratch. The most common technique is to implement the `BeanContextProxy` interface and return an instance of `BeanContextServicesSupport` from the `getBeanContextProxy()` method.



```
public class BeanContextServicesSupport extends BeanContextSupport implements BeanContextServices {
    // Public Constructors
    public BeanContextServicesSupport();
    public BeanContextServicesSupport(BeanContextServices peer);
    public BeanContextServicesSupport(BeanContextServices peer, java.util.Locale lc);
    public BeanContextServicesSupport(BeanContextServices peer, java.util.Locale lc, boolean dtime);
    public BeanContextServicesSupport(BeanContextServices peer, java.util.Locale lc, boolean dTime,
                                     boolean visible);

    // Inner Classes
    protected class BCSSChild extends BeanContextSupport.BCSCChild;
    protected class BCSSProxyServiceProvider implements BeanContextServiceProvider,
        BeanContextServiceRevokedListener;
    protected static class BCSSServiceProvider implements Serializable;

    // Protected Class Methods
    protected static final BeanContextServicesListener getChildBeanContextServicesListener(Object child);

    // Event Registration Methods (by event name)
    public void addBeanContextServicesListener(BeanContextServicesListener bcs);
    public void removeBeanContextServicesListener(BeanContextServicesListener bcs);

    // Public Instance Methods
    public BeanContextServices getBeanContextServicesPeer();

    // Methods Implementing BeanContextServiceRevokedListener
    public void serviceRevoked(BeanContextServiceRevokedEvent bcssre);

    // Methods Implementing BeanContextServices
    public void addBeanContextServicesListener(BeanContextServicesListener bcs);
}
```

## BeanContextServicesSupport

```
public boolean addService(Class serviceClass, BeanContextServiceProvider bcsp);
public java.util.Iterator getCurrentServiceClasses(); default: BeanContextSupport.BCSIterator
public java.util.Iterator getCurrentServiceSelectors(Class serviceClass);
public Object getService(BeanContextChild child, Object requestor, Class serviceClass, Object serviceSelector,
    BeanContextServiceRevokedListener bcsl) throws java.util TooManyListenersException;
public boolean hasService(Class serviceClass); synchronized
public void releaseService(BeanContextChild child, Object requestor, Object service);
public void removeBeanContextServicesListener(BeanContextServicesListener bcsl);
public void revokeService(Class serviceClass, BeanContextServiceProvider bcsp,
    boolean revokeCurrentServicesNow);
// Methods Implementing BeanContextServicesListener
public void serviceAvailable(BeanContextServiceAvailableEvent bcssae);
// Public Methods Overriding BeanContextSupport
public void initialize();
// Protected Methods Overriding BeanContextSupport
protected void bcsPreDeserializationHook(java.io.ObjectInputStream ois) synchronized
    throws java.io.IOException, ClassNotFoundException;
protected void bcsPreSerializationHook(java.io.ObjectOutputStream oos) synchronized
    throws java.io.IOException;
protected void childJustRemovedHook(Object child, BeanContextSupport.BCSChild bcsc);
protected BeanContextSupport.BCSChild createBCSChild(Object targetChild, Object peer);
// Protected Methods Overriding BeanContextChildSupport
protected void initializeBeanContextResources(); synchronized
protected void releaseBeanContextResources(); synchronized
// Protected Instance Methods
protected boolean addService(Class serviceClass, BeanContextServiceProvider bcsp, boolean fireEvent);
protected BeanContextServicesSupport.BCSSServiceProvider createBCSSServiceProvider(Class sc,
    BeanContextServiceProvider bcsp);
protected final void fireServiceAdded(BeanContextServiceAvailableEvent bcssae);
protected final void fireServiceAdded(Class serviceClass);
protected final void fireServiceRevoked(BeanContextServiceRevokedEvent bcsl);
protected final void fireServiceRevoked(Class serviceClass, boolean revokeNow);
// Protected Instance Fields
protected transient java.util.ArrayList bcsListeners;
protected transient BeanContextServicesSupport.BCSSProxyServiceProvider proxy;
protected transient int serializable;
protected transient java.util.HashMap services;
}
```

### BeanContextServicesSupport.BCSSChild

Java 1.2

java.beans.beancontext

serializable

This class is used internally by BeanContextServicesSupport to associate additional information with each child of the bean context. It has no public or protected method or fields, but may be customized by subclassing.

```
protected class BeanContextServicesSupport.BCSSChild extends BeanContextSupport.BCSChild {
    // No Constructor
}
```

### BeanContextServicesSupport.BCSSProxyServiceProvider

Java 1.2

java.beans.beancontext

This inner class is used internally by BeanContextServicesSupport to properly handle delegation to the services provided by containing bean contexts. It implements the BeanCon-

textServiceProvider interface in terms of the methods of a containing BeanContextServices object.

```
protected class BeanContextServicesSupport.BCSSProxyServiceProvider
    implements BeanContextServiceProvider, BeanContextServiceRevokedListener {
    // No Constructor
    // Methods Implementing BeanContextServiceProvider
    public java.util.Iterator getCurrentServiceSelectors(BeanContextServices bcs, Class serviceClass);
    public Object getService(BeanContextServices bcs, Object requestor, Class serviceClass, Object serviceSelector);
    public void releaseService(BeanContextServices bcs, Object requestor, Object service);
    // Methods Implementing BeanContextServiceRevokedListener
    public void serviceRevoked(BeanContextServiceRevokedEvent bcsre);
}
```

Type Of: BeanContextServicesSupport.proxy

### BeanContextServicesSupport.BCSSServiceProvider

Java 1.2

java.beans.beancontext

serializable

This inner class is a trivial wrapper around a BeanContextServiceProvider object. Subclasses that want to associate additional information with each service provider can subclass this class and override the createBCSSServiceProvider() method of BeanContextServicesSupport.

```
protected static class BeanContextServicesSupport.BCSSServiceProvider implements Serializable {
    // No Constructor
    // Protected Instance Methods
    protected BeanContextServiceProvider getServiceProvider();
    // Protected Instance Fields
    protected BeanContextServiceProvider serviceProvider;
}
```

Returned By: BeanContextServicesSupport.createBCSSServiceProvider()

### BeanContextSupport

Java 1.2

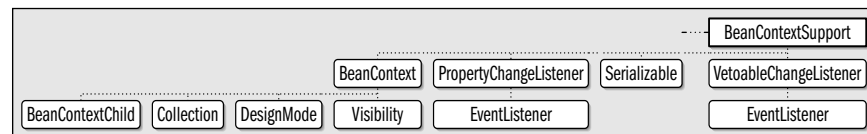
java.beans.beancontext

serializable collection

This class provides a simple, easily customizable implementation of BeanContext. Most bean context implementors find it easier to subclass BeanContextSupport or create a BeanContextSupport delegate object rather than implement the BeanContext interface from scratch.

Bean contexts are often AWT or Swing containers and cannot (because of a method-naming conflict) implement BeanContext. Therefore, a context object implements the BeanContextProxy interface and returns a BeanContext object from its getBeanContextProxy() method. A BeanContextSupport object is a suitable object to return from this method.

Some bean contexts require customized behavior, however, and BeanContextSupport is designed to be easily customized through subclassing. Protected methods such as child-JustAddedHook() and validatePendingAdd() are particularly useful when subclassing.



```
public class BeanContextSupport extends BeanContextChildSupport implements BeanContext,
    java.beans.PropertyChangeListener, Serializable, java.beans.VetoableChangeListener {
```

## *BeanContextSupport*

```
// Public Constructors
public BeanContextSupport();
public BeanContextSupport(BeanContext peer);
public BeanContextSupport(BeanContext peer, java.util.Locale lcle);
public BeanContextSupport(BeanContext peer, java.util.Locale lcle, boolean dtime);
public BeanContextSupport(BeanContext peer, java.util.Locale lcle, boolean dTime, boolean visible);

// Inner Classes
protected class BCSChild implements Serializable;
protected static final class BCSIterator implements java.util.Iterator;

// Protected Class Methods
protected static final boolean classEquals(Class first, Class second);
protected static final BeanContextChild getChildBeanContextChild(Object child);
protected static final BeanContextMembershipListener getChildBeanContextMembershipListener(
    Object child);
protected static final java.beans.PropertyChangeListener getChildPropertyChangeListener(Object child);
protected static final Serializable getChildSerializable(Object child);
protected static final java.beans.VetoableChangeListener getChildVetoableChangeListener(Object child);
protected static final java.beans.Visibility getChildVisibility(Object child);

// Event Registration Methods (by event name)
public void addBeanContextMembershipListener(                                Implements:BeanContext
    BeanContextMembershipListener bcml);
public void removeBeanContextMembershipListener(                            Implements:BeanContext
    BeanContextMembershipListener bcml);

// Property Accessor Methods (by property name)
public BeanContext getBeanContextPeer();                                default:BeanContextSupport
public boolean isDesignTime();                                           Implements:DesignMode synchronized default:false
public void setDesignTime(boolean dTime);                               Implements:DesignMode synchronized
public boolean isEmpty();                                               Implements:Collection default:true
public java.util.Locale getLocale();                                     synchronized
public void setLocale(java.util.Locale newLocale) throws java.beans.PropertyVetoException; synchronized
public boolean isSerializing();                                         default:false

// Public Instance Methods
public boolean containsKey(Object o);
public final void readChildren(java.io.ObjectInputStream ois) throws java.io.IOException, ClassNotFoundException;
public final void writeChildren(java.io.ObjectOutputStream oos) throws java.io.IOException;

// Methods Implementing BeanContext
public void addBeanContextMembershipListener(BeanContextMembershipListener bcml);
public java.net.URL getResource(String name, BeanContextChild bcc);
public java.io.InputStream getResourceAsStream(String name, BeanContextChild bcc);
public Object instantiateChild(String beanName) throws java.io.IOException, ClassNotFoundException;
public void removeBeanContextMembershipListener(BeanContextMembershipListener bcml);

// Methods Implementing Collection
public boolean add(Object targetChild);
public boolean addAll(java.util.Collection c);
public void clear();
public boolean contains(Object o);
public boolean containsAll(java.util.Collection c);
public boolean isEmpty();                                               default:true
public java.util.Iterator iterator();
public boolean remove(Object targetChild);
public boolean removeAll(java.util.Collection c);
public boolean retainAll(java.util.Collection c);
public int size();
public Object[] toArray();
public Object[] toArray(Object[] arry);
```

```
// Methods Implementing DesignMode
public boolean isDesignTime(); synchronized default:false
public void setDesignTime(boolean dTime); synchronized
// Methods Implementing PropertyChangeListener
public void propertyChange(java.beans.PropertyChangeEvent pce);
// Methods Implementing VetoableChangeListener
public void vetoableChange(java.beans.PropertyChangeEvent pce) throws java.beans.PropertyVetoException;
// Methods Implementing Visibility
public boolean avoidingGui();
public void dontUseGui(); synchronized
public boolean needsGui(); synchronized
public void okToUseGui(); synchronized
// Protected Instance Methods
protected java.util.Iterator bcsChildren();
protected void bcsPreDeserializationHook(java.io.ObjectInputStream ois) throws java.io.IOException, empty
    ClassNotFoundException;
protected void bcsPreSerializationHook(java.io.ObjectOutputStream oos) throws java.io.IOException; empty
protected void childDeserializedHook(Object child, BeanContextSupport.BCSCChild bcsc);
protected void childJustAddedHook(Object child, BeanContextSupport.BCSCChild bcsc); empty
protected void childJustRemovedHook(Object child, BeanContextSupport.BCSCChild bcsc); empty
protected final Object[] copyChildren();
protected BeanContextSupport.BCSCChild createBCSCChild(Object targetChild, Object peer);
protected final void deserialize(java.io.ObjectInputStream ois, java.util.Collection coll) throws java.io.IOException,
    ClassNotFoundException;
protected final void fireChildrenAdded(BeanContextMembershipEvent bcme);
protected final void fireChildrenRemoved(BeanContextMembershipEvent bcme);
protected void initialize(); synchronized
protected boolean remove(Object targetChild, boolean callChildSetBC);
protected final void serialize(java.io.ObjectOutputStream oos, java.util.Collection coll) throws java.io.IOException;
protected boolean validatePendingAdd(Object targetChild); constant
protected boolean validatePendingRemove(Object targetChild); constant
// Protected Instance Fields
protected transient java.util.ArrayList bcmListeners;
protected transient java.util.HashMap children;
protected boolean designTime;
protected java.util.Locale locale;
protected boolean okToUseGui;
}
```

*Subclasses:* BeanContextServicesSupport

## BeanContextSupport.BCSCChild

Java 1.2

java.beans.beancontext

serializable

This class is used internally by BeanContextSupport to keep track of additional information about its children. In particular, for children that implement the BeanContextProxy interface, it keeps track of the BeanContextChild object associated with the child. This class does not define any public or protected fields or methods. BeanContextSupport subclasses that want to associate additional information with each child can subclass this class and override the createBCSCChild() method to instantiate the new subclass.

```
protected class BeanContextSupport.BCSCChild implements Serializable {
// No Constructor
}
```

*Subclasses:* BeanContextServicesSupport.BCSCChild



### *BeanContextSupport.BCSCChild*

*Passed To:* BeanContextServicesSupport.childJustRemovedHook(),  
BeanContextSupport.{childDeserializedHook(), childJustAddedHook(), childJustRemovedHook()}

*Returned By:* BeanContextServicesSupport.createBCSCChild(),  
BeanContextSupport.createBCSCChild()

### **BeanContextSupport.BCSIterator**

**Java 1.2**

**java.beans.beancontext**

This class implements the `java.util.Iterator` interface. An instance of this class is returned by the `iterator()` method implemented by `BeanContextSupport`. The `remove()` method has an empty implementation and does not actually remove a child of the bean context.

```
protected static final class BeanContextSupport.BCSIterator implements java.util.Iterator {  
    // No Constructor  
    // Methods Implementing Iterator  
    public boolean hasNext();  
    public Object next();  
    public void remove();  
}
```

*empty*