JavaScript is disabled on your browser.

[Skip navigation links](#1fob9te)

* [Overview](http://docs.google.com/overview-summary.html)
* Package
* Class
* [Use](http://docs.google.com/package-use.html)
* [Tree](http://docs.google.com/package-tree.html)
* [Deprecated](http://docs.google.com/deprecated-list.html)
* [Index](http://docs.google.com/index-all.html)
* [Help](http://docs.google.com/help-doc.html)
* Prev Package
* [Next Package](http://docs.google.com/org/apache/commons/beanutils/converters/package-summary.html)
* [Frames](http://docs.google.com/index.html?org/apache/commons/beanutils/package-summary.html)
* [No Frames](http://docs.google.com/package-summary.html)
* [All Classes](http://docs.google.com/allclasses-noframe.html)

Package org.apache.commons.beanutils

The *Bean Introspection Utilities* component of the Apache Commons subproject offers low-level utility classes that assist in getting and setting property values on Java classes that follow the naming design patterns outlined in the JavaBeans Specification, as well as mechanisms for dynamically defining and accessing bean properties.

See: [Description](#3znysh7)

* Interface Summary

|  |  |
| --- | --- |
| * Interface | * Description |
| * [BeanIntrospector](http://docs.google.com/org/apache/commons/beanutils/BeanIntrospector.html) | * Definition of an interface for components that can perform introspection on bean classes. |
| * [Converter](http://docs.google.com/org/apache/commons/beanutils/Converter.html) | * General purpose data type converter that can be registered and used within the BeanUtils package to manage the conversion of objects from one type to another. |
| * [DynaBean](http://docs.google.com/org/apache/commons/beanutils/DynaBean.html) | * A **DynaBean** is a Java object that supports properties whose names and data types, as well as values, may be dynamically modified. |
| * [DynaClass](http://docs.google.com/org/apache/commons/beanutils/DynaClass.html) | * A **DynaClass** is a simulation of the functionality of java.lang.Class for classes implementing the DynaBean interface. |
| * [IntrospectionContext](http://docs.google.com/org/apache/commons/beanutils/IntrospectionContext.html) | * A context interface used during introspection for querying and setting property descriptors. |
| * [MutableDynaClass](http://docs.google.com/org/apache/commons/beanutils/MutableDynaClass.html) | * A specialized extension to DynaClass that allows properties to be added or removed dynamically. |

* Class Summary

|  |  |
| --- | --- |
| * Class | * Description |
| * [BaseDynaBeanMapDecorator](http://docs.google.com/org/apache/commons/beanutils/BaseDynaBeanMapDecorator.html)<K> | * A base class for decorators providing Map behavior on [DynaBean](http://docs.google.com/org/apache/commons/beanutils/DynaBean.html)s. |
| * [BasicDynaBean](http://docs.google.com/org/apache/commons/beanutils/BasicDynaBean.html) | * Minimal implementation of the DynaBean interface. |
| * [BasicDynaClass](http://docs.google.com/org/apache/commons/beanutils/BasicDynaClass.html) | * Minimal implementation of the DynaClass interface. |
| * [BeanComparator](http://docs.google.com/org/apache/commons/beanutils/BeanComparator.html)<T> | * This comparator compares two beans by the specified bean property. |
| * [BeanMap](http://docs.google.com/org/apache/commons/beanutils/BeanMap.html) | * An implementation of Map for JavaBeans which uses introspection to get and put properties in the bean. |
| * [BeanMap.Entry](http://docs.google.com/org/apache/commons/beanutils/BeanMap.Entry.html) | * Map entry used by [BeanMap](http://docs.google.com/org/apache/commons/beanutils/BeanMap.html). |
| * [BeanPredicate](http://docs.google.com/org/apache/commons/beanutils/BeanPredicate.html) | * Predicate implementation that applies the given Predicate to the result of calling the given property getter. |
| * [BeanPropertyValueChangeClosure](http://docs.google.com/org/apache/commons/beanutils/BeanPropertyValueChangeClosure.html) | * Closure that sets a property. |
| * [BeanPropertyValueEqualsPredicate](http://docs.google.com/org/apache/commons/beanutils/BeanPropertyValueEqualsPredicate.html) | * Predicate that evaluates a property value against a specified value. |
| * [BeanToPropertyValueTransformer](http://docs.google.com/org/apache/commons/beanutils/BeanToPropertyValueTransformer.html) | * Transformer that outputs a property value. |
| * [BeanUtils](http://docs.google.com/org/apache/commons/beanutils/BeanUtils.html) | * Utility methods for populating JavaBeans properties via reflection. |
| * [BeanUtilsBean](http://docs.google.com/org/apache/commons/beanutils/BeanUtilsBean.html) | * JavaBean property population methods. |
| * [BeanUtilsBean2](http://docs.google.com/org/apache/commons/beanutils/BeanUtilsBean2.html) | * [BeanUtilsBean](http://docs.google.com/org/apache/commons/beanutils/BeanUtilsBean.html) implementation that creates a [ConvertUtilsBean2](http://docs.google.com/org/apache/commons/beanutils/ConvertUtilsBean2.html) and delegates conversion to [ConvertUtilsBean.convert(Object, Class)](http://docs.google.com/org/apache/commons/beanutils/ConvertUtilsBean.html#convert-java.lang.Object-java.lang.Class-). |
| * [ConstructorUtils](http://docs.google.com/org/apache/commons/beanutils/ConstructorUtils.html) | * Utility reflection methods focussed on constructors, modelled after [MethodUtils](http://docs.google.com/org/apache/commons/beanutils/MethodUtils.html). |
| * [ContextClassLoaderLocal](http://docs.google.com/org/apache/commons/beanutils/ContextClassLoaderLocal.html)<T> | * An instance of this class represents a value that is provided per (thread) context classloader. |
| * [ConvertingWrapDynaBean](http://docs.google.com/org/apache/commons/beanutils/ConvertingWrapDynaBean.html) | * Implementation of DynaBean that wraps a standard JavaBean instance, so that DynaBean APIs can be used to access its properties, though this implementation allows type conversion to occur when properties are set. |
| * [ConvertUtils](http://docs.google.com/org/apache/commons/beanutils/ConvertUtils.html) | * Utility methods for converting String scalar values to objects of the specified Class, String arrays to arrays of the specified Class. |
| * [ConvertUtilsBean](http://docs.google.com/org/apache/commons/beanutils/ConvertUtilsBean.html) | * Utility methods for converting String scalar values to objects of the specified Class, String arrays to arrays of the specified Class. |
| * [ConvertUtilsBean2](http://docs.google.com/org/apache/commons/beanutils/ConvertUtilsBean2.html) | * [ConvertUtilsBean](http://docs.google.com/org/apache/commons/beanutils/ConvertUtilsBean.html) implementation that delegates convert() methods to the new [ConvertUtilsBean.convert(Object, Class)](http://docs.google.com/org/apache/commons/beanutils/ConvertUtilsBean.html#convert-java.lang.Object-java.lang.Class-) method. |
| * [DefaultBeanIntrospector](http://docs.google.com/org/apache/commons/beanutils/DefaultBeanIntrospector.html) | * The default [BeanIntrospector](http://docs.google.com/org/apache/commons/beanutils/BeanIntrospector.html) implementation. |
| * [DynaBeanMapDecorator](http://docs.google.com/org/apache/commons/beanutils/DynaBeanMapDecorator.html) | * Deprecated Use [DynaBeanPropertyMapDecorator](http://docs.google.com/org/apache/commons/beanutils/DynaBeanPropertyMapDecorator.html) instead. |
| * [DynaBeanPropertyMapDecorator](http://docs.google.com/org/apache/commons/beanutils/DynaBeanPropertyMapDecorator.html) | * Decorates a [DynaBean](http://docs.google.com/org/apache/commons/beanutils/DynaBean.html) to provide Map behavior. |
| * [DynaProperty](http://docs.google.com/org/apache/commons/beanutils/DynaProperty.html) | * The metadata describing an individual property of a DynaBean. |
| * [FluentPropertyBeanIntrospector](http://docs.google.com/org/apache/commons/beanutils/FluentPropertyBeanIntrospector.html) | * An implementation of the BeanIntrospector interface which can detect write methods for properties used in fluent API scenario. |
| * [LazyDynaBean](http://docs.google.com/org/apache/commons/beanutils/LazyDynaBean.html) | * DynaBean which automatically adds properties to the DynaClass and provides *Lazy List* and *Lazy Map* features. |
| * [LazyDynaClass](http://docs.google.com/org/apache/commons/beanutils/LazyDynaClass.html) | * DynaClass which implements the MutableDynaClass interface. |
| * [LazyDynaList](http://docs.google.com/org/apache/commons/beanutils/LazyDynaList.html) | * *Lazy* DynaBean List. |
| * [LazyDynaMap](http://docs.google.com/org/apache/commons/beanutils/LazyDynaMap.html) | * Provides a *light weight* DynaBean facade to a Map with *lazy* map/list processing. |
| * [MappedPropertyDescriptor](http://docs.google.com/org/apache/commons/beanutils/MappedPropertyDescriptor.html) | * A MappedPropertyDescriptor describes one mapped property. |
| * [MethodUtils](http://docs.google.com/org/apache/commons/beanutils/MethodUtils.html) | * Utility reflection methods focused on methods in general rather than properties in particular. |
| * [PropertyUtils](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html) | * Utility methods for using Java Reflection APIs to facilitate generic property getter and setter operations on Java objects. |
| * [PropertyUtilsBean](http://docs.google.com/org/apache/commons/beanutils/PropertyUtilsBean.html) | * Utility methods for using Java Reflection APIs to facilitate generic property getter and setter operations on Java objects. |
| * [ResultSetDynaClass](http://docs.google.com/org/apache/commons/beanutils/ResultSetDynaClass.html) | * Implementation of DynaClass for DynaBeans that wrap the java.sql.Row objects of a java.sql.ResultSet. |
| * [ResultSetIterator](http://docs.google.com/org/apache/commons/beanutils/ResultSetIterator.html) | * Implementation of java.util.Iterator returned by the iterator() method of [ResultSetDynaClass](http://docs.google.com/org/apache/commons/beanutils/ResultSetDynaClass.html). |
| * [RowSetDynaClass](http://docs.google.com/org/apache/commons/beanutils/RowSetDynaClass.html) | * Implementation of [DynaClass](http://docs.google.com/org/apache/commons/beanutils/DynaClass.html) that creates an in-memory collection of [DynaBean](http://docs.google.com/org/apache/commons/beanutils/DynaBean.html)s representing the results of an SQL query. |
| * [SuppressPropertiesBeanIntrospector](http://docs.google.com/org/apache/commons/beanutils/SuppressPropertiesBeanIntrospector.html) | * A specialized BeanIntrospector implementation which suppresses some properties. |
| * [WrapDynaBean](http://docs.google.com/org/apache/commons/beanutils/WrapDynaBean.html) | * Implementation of DynaBean that wraps a standard JavaBean instance, so that DynaBean APIs can be used to access its properties. |
| * [WrapDynaClass](http://docs.google.com/org/apache/commons/beanutils/WrapDynaClass.html) | * Implementation of DynaClass for DynaBeans that wrap standard JavaBean instances. |

* Exception Summary

|  |  |
| --- | --- |
| * Exception | * Description |
| * [BeanAccessLanguageException](http://docs.google.com/org/apache/commons/beanutils/BeanAccessLanguageException.html) | * Thrown to indicate that the *Bean Access Language* cannot execute query against given bean. |
| * [ConversionException](http://docs.google.com/org/apache/commons/beanutils/ConversionException.html) | * A **ConversionException** indicates that a call to Converter.convert() has failed to complete successfully. |
| * [NestedNullException](http://docs.google.com/org/apache/commons/beanutils/NestedNullException.html) | * Thrown to indicate that the *Bean Access Language* cannot execute query against given bean since a nested bean referenced is null. |

## Package org.apache.commons.beanutils Description

The *Bean Introspection Utilities* component of the Apache Commons subproject offers low-level utility classes that assist in getting and setting property values on Java classes that follow the naming design patterns outlined in the JavaBeans Specification, as well as mechanisms for dynamically defining and accessing bean properties.

# Table of Contents

* 1. [Overview](#2et92p0)
  + 1.1 [Background](#tyjcwt)
  + 1.2 [External Dependencies](#3dy6vkm)
* 2. [Standard JavaBeans](#1t3h5sf)
  + 2.1 [Background](#4d34og8)
  + 2.2 [Basic Property Access](#2s8eyo1)
  + 2.3 [Nested Property Access](#17dp8vu)
  + 2.4 [Customizing Introspection](#3rdcrjn)
  + 2.5 [Suppressing Properties](#26in1rg)
* 3. [Dynamic Beans (DynaBeans)](#lnxbz9)
  + 3.1 [Background](#35nkun2)
  + 3.2 [BasicDynaBean and BasicDynaClass](#1ksv4uv)
  + 3.3 [ResultSetDynaClass (Wraps ResultSet in DynaBeans)](#44sinio)
  + 3.4 [RowSetDynaClass (Disconnected ResultSet as DynaBeans)](#2jxsxqh)
  + 3.5 [WrapDynaBean and WrapDynaClass](#z337ya)
  + 3.6 [*Lazy* DynaBeans](#3j2qqm3)
* 4. [Data Type Conversions](#1y810tw)
  + 4.1 [Background](#4i7ojhp)
  + 4.2 [BeanUtils and ConvertUtils Conversions](#2xcytpi)
  + 4.3 [Defining Your Own Converters](#1ci93xb)
  + 4.4 [Locale Aware Conversions](#3whwml4)
* 5. [Utility Objects And Static Utility Classes](#2bn6wsx)
* 6. [Collections](#3as4poj)
  + 6.1 [Comparing Beans](#1pxezwc)
  + 6.2 [Operating On Collections Of Beans](#49x2ik5)
  + 6.3 [Querying Or Filtering Collections Of Beans](#2p2csry)
  + 6.4 [Transforming Collections Of Beans](#41mghml)
* 7. [Frequently Asked Questions](#147n2zr)
  + [Why Can't BeanUtils Find My Method?](#3o7alnk)
  + [How Do I Set The BeanComparator Order To Be Ascending/Descending?](#23ckvvd)

# 1. Overview

## 1.1 Background

The *JavaBeans* name comes from a [Java API](http://java.sun.com/products/javabeans/) for a component architecture for the Java language. Writing Java classes that conform to the JavaBeans design patterns makes it easier for Java developers to understand the functionality provided by your class, as well as allowing JavaBeans-aware tools to use Java's *introspection* capabilities to learn about the properties and operations provided by your class, and present them in a visually appealing manner in development tools.

The [JavaBeans Specification](http://java.sun.com/products/javabeans/docs/spec.html) describes the complete set of characteristics that makes an arbitrary Java class a JavaBean or not -- and you should consider reading this document to be an important part of developing your Java programming skills. However, the required characteristics of JavaBeans that are important for most development scenarios are listed here:

* The class must be **public**, and provide a **public** constructor that accepts no arguments. This allows tools and applications to dynamically create new instances of your bean, without necessarily knowing what Java class name will be used ahead of time, like this:  
   String className = ...;  
   Class beanClass = Class.forName(className);  
   Object beanInstance = beanClass.newInstance();
* As a necessary consequence of having a no-arguments constructor, configuration of your bean's behavior must be accomplished separately from its instantiation. This is typically done by defining a set of *properties* of your bean, which can be used to modify its behavior or the data that the bean represents. The normal convention for property names is that they start with a lower case letter, and be comprised only of characters that are legal in a Java identifier.
* Typically, each bean property will have a public *getter* and *setter* method that are used to retrieve or define the property's value, respectively. The JavaBeans Specification defines a design pattern for these names, using get or set as the prefix for the property name with it's first character capitalized. Thus, you a JavaBean representing an employee might have (among others) properties named firstName, lastName, and hireDate, with method signatures like this:  
   public class Employee {  
   public Employee(); // Zero-arguments constructor  
   public String getFirstName();  
   public void setFirstName(String firstName);  
   public String getLastName();  
   public void setLastName(String lastName);  
   public Date getHireDate();  
   public void setHireDate(Date hireDate);  
   public boolean isManager();  
   public void setManager(boolean manager);  
   public String getFullName();  
   }
* As you can see from the above example, there is a special variant allowed for boolean properties -- you can name the *getter* method with a is prefix instead of a get prefix if that makes for a more understandable method name.
* If you have both a *getter* and a *setter* method for a property, the data type returned by the *getter* must match the data type accepted by the *setter*. In addition, it is contrary to the JavaBeans specification to have more than one *setter* with the same name, but different property types.
* It is not required that you provide a *getter* and a *setter* for every property. In the example above, the fullName property is read-only, because there is no *setter* method. It is also possible, but less common, to provide write-only properties.
* It is also possible to create a JavaBean where the *getter* and *setter* methods do not match the naming pattern described above. The standard JavaBeans support classes in the Java language, as well as all classes in the BeanUtils package, allow you to describe the actual property method names in a BeanInfo class associated with your bean class. See the JavaBeans Specification for full details.
* The JavaBeans Specification also describes many additional design patterns for event listeners, wiring JavaBeans together into component hierarchies, and other useful features that are beyond the scope of the BeanUtils package.

Using standard Java coding techniques, it is very easy to deal with JavaBeans if you know ahead of time which bean classes you will be using, and which properties you are interested in:

Employee employee = ...;  
 System.out.println("Hello " + employee.getFirstName() + "!");

## 1.2 External Dependencies

The *commons-beanutils* package requires that the following additional packages be available in the application's class path at runtime:

* [Logging Package (Apache Commons)](http://commons.apache.org/downloads/download_logging.cgi), version 1.0 or later
* [Collections Package (Apache Commons)](http://commons.apache.org/downloads/download_collections.cgi), version 1.0 or later

# 2. Standard JavaBeans

## 2.1 Background

As described above, the standard facilities of the Java programming language make it easy and natural to access the property values of your beans using calls to the appropriate getter methods. But what happens in more sophisticated environments where you do not necessarily know ahead of time which bean class you are going to be using, or which property you want to retrieve or modify? The Java language provides classes like java.beans.Introspector, which can examine a Java class at runtime and identify for you the names of the property getter and setter methods, plus the *Reflection* capabilities to dynamically call such a method. However, these APIs can be difficult to use, and expose the application developer to many unnecessary details of the underlying structure of Java classes. The APIs in the BeanUtils package are intended to simplify getting and setting bean properties dynamically, where the objects you are accessing -- and the names of the properties you care about -- are determined at runtime in your application, rather than as you are writing and compiling your application's classes.

This is the set of needs that are satisfied by the static methods of the [PropertyUtils](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html) class, which are described further in this section. First, however, some further definitions will prove to be useful:

The general set of possible property types supported by a JavaBean can be broken into three categories -- some of which are supported by the standard JavaBeans specification, and some of which are uniquely supported by the *BeanUtils* package:

* **Simple** - Simple, or scalar, properties have a single value that may be retrieved or modified. The underlying property type might be a Java language primitive (such as int, a simple object (such as a java.lang.String), or a more complex object whose class is defined either by the Java language, by the application, or by a class library included with the application.
* **Indexed** - An indexed property stores an ordered collection of objects (all of the same type) that can be individually accessed by an integer-valued, non-negative index (or subscript). Alternatively, the entire set of values may be set or retrieved using an array. As an extension to the JavaBeans specification, the *BeanUtils* package considers any property whose underlying data type is java.util.List (or an implementation of List) to be indexed as well.
* **Mapped** - As an extension to standard JavaBeans APIs, the *BeanUtils* package considers any property whose underlying value is a java.util.Map to be "mapped". You can set and retrieve individual values via a String-valued key.

A variety of API methods are provided in the [PropertyUtils](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html) class to get and set property values of all of these types. In the code fragments below, assume that there are two bean classes defined with the following method signatures:

public class Employee {  
 public Address getAddress(String type);  
 public void setAddress(String type, Address address);  
 public Employee getSubordinate(int index);  
 public void setSubordinate(int index, Employee subordinate);  
 public String getFirstName();  
 public void setFirstName(String firstName);  
 public String getLastName();  
 public void setLastName(String lastName);  
 }

## 2.2 Basic Property Access

Getting and setting **simple** property values is, well, simple :-). Check out the following API signatures in the Javadocs:

* [PropertyUtils.getSimpleProperty(Object, String)](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html#getSimpleProperty-java.lang.Object-java.lang.String-)
* [PropertyUtils.setSimpleProperty(Object, String, Object)](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html#setSimpleProperty-java.lang.Object-java.lang.String-java.lang.Object-)

Using these methods, you might dynamically manipulate the employee's name in an application:

Employee employee = ...;  
 String firstName = (String)  
 PropertyUtils.getSimpleProperty(employee, "firstName");  
 String lastName = (String)  
 PropertyUtils.getSimpleProperty(employee, "lastName");  
 ... manipulate the values ...  
 PropertyUtils.setSimpleProperty(employee, "firstName", firstName);  
 PropertyUtils.setSimpleProperty(employee, "lastName", lastName);

For **indexed** properties, you have two choices - you can either build a subscript into the "property name" string, using square brackets, or you can specify the subscript in a separate argument to the method call:

* [PropertyUtils.getIndexedProperty(Object, String)](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html#getIndexedProperty-java.lang.Object-java.lang.String-)
* [PropertyUtils.getIndexedProperty(Object, String, int)](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html#getIndexedProperty-java.lang.Object-java.lang.String-int-)
* [PropertyUtils.setIndexedProperty(Object, String, Object)](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html#setIndexedProperty-java.lang.Object-java.lang.String-java.lang.Object-)
* [PropertyUtils.setIndexedProperty(Object, String, int, Object)](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html#setIndexedProperty-java.lang.Object-java.lang.String-int-java.lang.Object-)

Only integer constants are allowed when you add a subscript to the property name. If you need to calculate the index of the entry you wish to retrieve, you can use String concatenation to assemble the property name expression. For example, you might do either of the following:

Employee employee = ...;  
 int index = ...;  
 String name = "subordinate[" + index + "]";  
 Employee subordinate = (Employee)  
 PropertyUtils.getIndexedProperty(employee, name);  
  
 Employee employee = ...;  
 int index = ...;  
 Employee subordinate = (Employee)  
 PropertyUtils.getIndexedProperty(employee, "subordinate", index);

In a similar manner, there are two possible method signatures for getting and setting **mapped** properties. The difference is that the extra argument is surrounded by parentheses ("(" and ")") instead of square brackets, and it is considered to be a String-value key used to get or set the appropriate value from an underlying map.

* [PropertyUtils.getMappedProperty(Object, String)](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html#getMappedProperty-java.lang.Object-java.lang.String-)
* [PropertyUtils.getMappedProperty(Object, String, String)](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html#getMappedProperty-java.lang.Object-java.lang.String-java.lang.String-)
* [PropertyUtils.setMappedProperty(Object, String, Object)](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html#setMappedProperty-java.lang.Object-java.lang.String-java.lang.Object-)
* [PropertyUtils.setMappedProperty(Object, String, String, Object)](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html#setMappedProperty-java.lang.Object-java.lang.String-java.lang.String-java.lang.Object-)

You can, for example, set the employee's home address in either of these two manners:

Employee employee = ...;  
 Address address = ...;  
 PropertyUtils.setMappedProperty(employee, "address(home)", address);  
  
 Employee employee = ...;  
 Address address = ...;  
 PropertyUtils.setMappedProperty(employee, "address", "home", address);

## 2.3 Nested Property Access

In all of the examples above, we have assumed that you wished to retrieve the value of a property of the bean being passed as the first argument to a PropertyUtils method. However, what if the property value you retrieve is really a Java object, and you wish to retrieve a property of *that* object instead?

For example, assume we really wanted the city property of the employee's home address. Using standard Java programming techniques for direct access to the bean properties, we might write:

String city = employee.getAddress("home").getCity();

The equivalent mechanism using the PropertyUtils class is called **nested** property access. To use this approach, you concatenate together the property names of the access path, using "." separators -- very similar to the way you can perform nested property access in JavaScript.

* [PropertyUtils.getNestedProperty(Object, String)](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html#getNestedProperty-java.lang.Object-java.lang.String-)
* [PropertyUtils.setNestedProperty(Object, String, Object)](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html#setNestedProperty-java.lang.Object-java.lang.String-java.lang.Object-)

The PropertyUtils equivalent to the above Java expression would be:

String city = (String)  
 PropertyUtils.getNestedProperty(employee, "address(home).city");

Finally, for convenience, PropertyUtils provides method signatures that accept any arbitrary combination of simple, indexed, and mapped property access, using any arbitrary level of nesting:

* [PropertyUtils.getProperty(Object, String)](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html#getProperty-java.lang.Object-java.lang.String-)
* [PropertyUtils.setProperty(Object, String, Object)](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html#setProperty-java.lang.Object-java.lang.String-java.lang.Object-)

which you might use like this:

Employee employee = ...;  
 String city = (String) PropertyUtils.getProperty(employee,  
 "subordinate[3].address(home).city");

## 2.4 Customizing Introspection

As was pointed out, BeanUtils relies on conventions defined by the *JavaBeans* specification to determine the properties available for a specific bean class. Thus all classes conforming to these conventions can be used out of the box.

Sometimes an application has to deal with classes using different conventions. For instance, fluent APIs allowing method chaining are not compliant to the *JavaBeans* specification because here set methods have non-void return values. From version 1.9.0 onwards, BeanUtils supports customization of its introspection mechanism. This allows an application to extend or modify the default discovery of bean properties.

The key to this extension mechanism is the [BeanIntrospector](http://docs.google.com/org/apache/commons/beanutils/BeanIntrospector.html) interface. The purpose of an object implementing this interface is to process a specific target class and create corresponding PropertyDescriptor objects for the properties it detects. Per default, BeanUtils uses a [DefaultBeanIntrospector](http://docs.google.com/org/apache/commons/beanutils/DefaultBeanIntrospector.html) object which detects properties compatible with the *JavaBeans* specification.

In order to extend the property discovery mechanism, PropertyUtils offers the [PropertyUtils.addBeanIntrospector(BeanIntrospector)](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html#addBeanIntrospector-org.apache.commons.beanutils.BeanIntrospector-) method. Here a custom BeanIntrospector implementation can be passed in. During introspection of a class, this custom introspector is then called, and it can add the properties it has detected to the total result. As an example of such a custom BeanIntrospector implementation, BeanUtils ships with the [FluentPropertyBeanIntrospector](http://docs.google.com/org/apache/commons/beanutils/FluentPropertyBeanIntrospector.html) class. This implementation can detect properties whose set methods have a non-void return type - thus enabling support for typical properties in a fluent API.

## 2.5 Suppressing Properties

The mechanism of customizing bean introspection described in the previous section can also be used to suppress specific properties. There is a specialized BeanIntrospector implementation that does exactly this: [SuppressPropertiesBeanIntrospector](http://docs.google.com/org/apache/commons/beanutils/SuppressPropertiesBeanIntrospector.html). When creating an instance, a collection with the names of properties that should not be accessible on beans has to be provided. These properties will then be removed if they have been detected by other BeanIntrospector instances during processing of a bean class.

A good use case for suppressing properties is the special class property which is per default available for all beans; it is generated from the getClass() method inherited from Object which follows the naming conventions for property get methods. Exposing this property in an uncontrolled way can lead to a security vulnerability as it allows access to the class loader. More information can be found at <https://issues.apache.org/jira/browse/BEANUTILS-463>.

Because the class property is undesired in many use cases there is already an instance of SuppressPropertiesBeanIntrospector which is configured to suppress this property. It can be obtained via the SUPPRESS\_CLASS constant of SuppressPropertiesBeanIntrospector.

# 3. Dynamic Beans (DynaBeans)

## 3.1 Background

The [PropertyUtils](http://docs.google.com/org/apache/commons/beanutils/PropertyUtils.html) class described in the preceding section is designed to provide dynamic property access on existing JavaBean classes, without modifying them in any way. A different use case for dynamic property access is when you wish to represent a dynamically calculated set of property values as a JavaBean, but *without* having to actually write a Java class to represent these properties. Besides the effort savings in not having to create and maintain a separate Java class, this ability also means you can deal with situations where the set of properties you care about is determined dynamically (think of representing the result set of an SQL select as a set of JavaBeans ...).

To support this use case, the *BeanUtils* package provides the [DynaBean](http://docs.google.com/org/apache/commons/beanutils/DynaBean.html) interface, which must be implemented by a bean class actually implementing the interface's methods, and the associated [DynaClass](http://docs.google.com/org/apache/commons/beanutils/DynaClass.html) interface that defines the set of properties supported by a particular group of DynaBeans, in much the same way that java.lang.Class defines the set of properties supported by all instances of a particular JavaBean class.

For example, the Employee class used in the examples above might be implemented as a DynaBean, rather than as a standard JavaBean. You can access its properties like this:

DynaBean employee = ...; // Details depend on which  
 // DynaBean implementation you use  
 String firstName = (String) employee.get("firstName");  
 Address homeAddress = (Address) employee.get("address", "home");  
 Object subordinate = employee.get("subordinate", 2);

One very important convenience feature should be noted: *the PropertyUtils property getter and setter methods understand how to access properties in DynaBeans*. Therefore, if the bean you pass as the first argument to, say, PropertyUtils.getSimpleProperty() is really a DynaBean implementation, the call will get converted to the appropriate DynaBean getter method transparently. Thus, you can base your application's dynamic property access totally on the PropertyUtils APIs, if you wish, and use them to access either standard JavaBeans or DynaBeans without having to care ahead of time how a particular bean is implemented.

Because DynaBean and DynaClass are interfaces, they may be implemented multiple times, in different ways, to address different usage scenarios. The following subsections describe the implementations that are provided as a part of the standard *BeanUtils* package, although you are encouraged to provide your own custom implementations for cases where the standard implementations are not sufficient.

## 3.2 BasicDynaBean and BasicDynaClass

The [BasicDynaBean](http://docs.google.com/org/apache/commons/beanutils/BasicDynaBean.html) and [BasicDynaClass](http://docs.google.com/org/apache/commons/beanutils/BasicDynaClass.html) implementation provides a basic set of dynamic property capabilities where you want to dynamically define the set of properties (described by instances of [DynaProperty](http://docs.google.com/org/apache/commons/beanutils/DynaProperty.html)). You start by defining the DynaClass that establishes the set of properties you care about:

DynaProperty[] props = new DynaProperty[]{  
 new DynaProperty("address", java.util.Map.class),  
 new DynaProperty("subordinate", mypackage.Employee[].class),  
 new DynaProperty("firstName", String.class),  
 new DynaProperty("lastName", String.class)  
 };  
 BasicDynaClass dynaClass = new BasicDynaClass("employee", null, props);

Note that the 'dynaBeanClass' argument (in the constructor of BasicDynaClass) can have the value of null. In this case, the value of dynaClass.getDynaBeanClass will just be the Class for BasicDynaBean.

Next, you use the newInstance() method of this DynaClass to create new DynaBean instances that conform to this DynaClass, and populate its initial property values (much as you would instantiate a new standard JavaBean and then call its property setters):

DynaBean employee = dynaClass.newInstance();  
 employee.set("address", new HashMap());  
 employee.set("subordinate", new mypackage.Employee[0]);  
 employee.set("firstName", "Fred");  
 employee.set("lastName", "Flintstone");

Note that the DynaBean class was declared to be DynaBean instead of BasicDynaBean. In general, if you are using DynaBeans, you will not want to care about the actual implementation class that is being used -- you only care about declaring that it is a DynaBean so that you can use the DynaBean APIs.

As stated above, you can pass a DynaBean instance as the first argument to a PropertyUtils method that gets and sets properties, and it will be interpreted as you expect -- the dynamic properties of the DynaBean will be retrieved or modified, instead of underlying properties on the actual BasicDynaBean implementation class.

## 3.3 ResultSetDynaClass (Wraps ResultSet in DynaBeans)

A very common use case for DynaBean APIs is to wrap other collections of "stuff" that do not normally present themselves as JavaBeans. One of the most common collections that would be nice to wrap is the java.sql.ResultSet that is returned when you ask a JDBC driver to perform a SQL SELECT statement. Commons BeanUtils offers a standard mechanism for making each row of the result set visible as a DynaBean, which you can utilize as shown in this example:

Connection conn = ...;  
 Statement stmt = conn.createStatement();  
 ResultSet rs = stmt.executeQuery  
 ("select account\_id, name from customers");  
 Iterator rows = (new ResultSetDynaClass(rs)).iterator();  
 while (rows.hasNext()) {  
 DynaBean row = (DynaBean) rows.next();  
 System.out.println("Account number is " +  
 row.get("account\_id") +  
 " and name is " + row.get("name"));  
 }  
 rs.close();  
 stmt.close();

## 3.4 RowSetDynaClass (Disconnected ResultSet as DynaBeans)

Although [ResultSetDynaClass](#44sinio) is a very useful technique for representing the results of an SQL query as a series of DynaBeans, an important problem is that the underlying ResultSet must remain open throughout the period of time that the rows are being processed by your application. This hinders the ability to use ResultSetDynaClass as a means of communicating information from the model layer to the view layer in a model-view-controller architecture such as that provided by the [Struts Framework](http://struts.apache.org/), because there is no easy mechanism to assure that the result set is finally closed (and the underlying Connection returned to its connection pool, if you are using one).

The RowSetDynaClass class represents a different approach to this problem. When you construct such an instance, the underlying data is *copied* into a set of in-memory DynaBeans that represent the result. The advantage of this technique, of course, is that you can immediately close the ResultSet (and the corresponding Statement), normally before you even process the actual data that was returned. The disadvantage, of course, is that you must pay the performance and memory costs of copying the result data, and the result data must fit entirely into available heap memory. For many environments (particularly in web applications), this tradeoff is usually quite beneficial.

As an additional benefit, the RowSetDynaClass class is defined to implement java.io.Serializable, so that it (and the DynaBeans that correspond to each row of the result) can be conveniently serialized and deserialized (as long as the underlying column values are also Serializable). Thus, RowSetDynaClass represents a very convenient way to transmit the results of an SQL query to a remote Java-based client application (such as an applet).

The normal usage pattern for a RowSetDynaClass will look something like this:

Connection conn = ...; // Acquire connection from pool  
 Statement stmt = conn.createStatement();  
 ResultSet rs = stmt.executeQuery("SELECT ...");  
 RowSetDynaClass rsdc = new RowSetDynaClass(rs);  
 rs.close();  
 stmt.close();  
 ...; // Return connection to pool  
 List rows = rsdc.getRows();  
 ...; // Process the rows as desired

## 3.5 WrapDynaBean and WrapDynaClass

OK, you've tried the DynaBeans APIs and they are cool -- very simple get() and set() methods provide easy access to all of the dynamically defined simple, indexed, and mapped properties of your DynaBeans. You'd like to use the DynaBean APIs to access **all** of your beans, but you've got a bunch of existing standard JavaBeans classes to deal with as well. This is where the [WrapDynaBean](http://docs.google.com/org/apache/commons/beanutils/WrapDynaBean.html) (and its associated [WrapDynaClass](http://docs.google.com/org/apache/commons/beanutils/WrapDynaClass.html)) come into play. As the name implies, a WrapDynaBean is used to "wrap" the DynaBean APIs around an existing standard JavaBean class. To use it, simply create the wrapper like this:

MyBean bean = ...;  
 DynaBean wrapper = new WrapDynaBean(bean);  
 String firstName = wrapper.get("firstName");

Note that, although appropriate WrapDynaClass instances are created internally, you never need to deal with them.

## 3.6 ***Lazy*** DynaBeans

* 1. [LazyDynaBean](#2grqrue) - A *Lazy* [DynaBean](http://docs.google.com/org/apache/commons/beanutils/DynaBean.html)
* 2. [LazyDynaMap](#vx1227) - A *light weight* [DynaBean](http://docs.google.com/org/apache/commons/beanutils/DynaBean.html) facade to a Map with *lazy* map/list processing
* 3. [LazyDynaList](#3fwokq0) - A *lazy list* for [DynaBean's](http://docs.google.com/org/apache/commons/beanutils/DynaBean.html), java.util.Map's or POJO beans.
* 4. [LazyDynaClass](#1v1yuxt) - A [MutableDynaClass](http://docs.google.com/org/apache/commons/beanutils/MutableDynaClass.html) implementation.

You bought into the DynaBeans because it saves coding all those POJO JavaBeans but you're here because *lazy* caught your eye and wondered whats that about? What makes these flavors of DynaBean *lazy* are the following features:

* ***Lazy* property addition** - lazy beans use a [DynaClass](http://docs.google.com/org/apache/commons/beanutils/DynaClass.html) which implements the [MutableDynaClass](http://docs.google.com/org/apache/commons/beanutils/MutableDynaClass.html) interface. This provides the ability to add and remove a DynaClass's properties. *Lazy* beans use this feature to automatically add a property which doesn't exist to the DynaClass when the set(name, value) method is called.
* ***Lazy* List/Array growth** - If an *indexed* property is not large enough to accomodate the index being set then the List or Array is automatically *grown* so that it is.
* ***Lazy* List/Array instantiation** - if an *indexed* property doesn't exist then calling the [DynaBean's](http://docs.google.com/org/apache/commons/beanutils/DynaBean.html) *indexed* property getter/setter methods (i.e. get(name, index) or set(name, index, value)) results in either a new List or Array being instantiated. If the indexed property has not been defined in the DynaClass then it is automatically added and a default List implementation instantiated.
* ***Lazy* Map instantiation** - if a *mapped* property doesn't exist then calling the [DynaBean's](http://docs.google.com/org/apache/commons/beanutils/DynaBean.html) *mapped* property getter/setter methods (i.e. get(name, key) or set(name, key, value)) results in a new Map being instantiated. If the mapped property has not been defined in the DynaClass then it is automatically added and a default Map implementation instantiated.
* ***Lazy* Bean instantiation** - if a property is defined in the DynaClass as a DynaBean or regular bean and doesn't exist in the DynaBean then LazyDynaBean wiill try to instantiate the bean using a default empty constructor.

**1.** [**LazyDynaBean**](http://docs.google.com/org/apache/commons/beanutils/LazyDynaBean.html) is the standard *lazy* bean implementation. By default it is associated with a [LazyDynaClass](http://docs.google.com/org/apache/commons/beanutils/LazyDynaClass.html) which implements the [MutableDynaClass](http://docs.google.com/org/apache/commons/beanutils/MutableDynaClass.html) interface - however it can be used with any MutableDynaClass implementation. The question is *how do I use it?* - well it can be as simple as creating a new bean and then calling the getters/setters...

DynaBean dynaBean = new LazyDynaBean();  
  
 dynaBean.set("foo", "bar"); // simple  
  
 dynaBean.set("customer", "title", "Mr"); // mapped  
 dynaBean.set("customer", "surname", "Smith"); // mapped  
  
 dynaBean.set("address", 0, addressLine1); // indexed  
 dynaBean.set("address", 1, addressLine2); // indexed  
 dynaBean.set("address", 2, addressLine3); // indexed

**2.** [**LazyDynaMap**](http://docs.google.com/org/apache/commons/beanutils/LazyDynaMap.html) is a *light weight* DynaBean facade to a Map with all the usual *lazy* features. Its *light weight* because it doesn't have an associated DynaClass containing all the properties. In fact it actually implements the DynaClass interface itself (and MutableDynaClass) and derives all the *DynaClass* information from the actual contents of the Map. A LazyDynaMap can be created around an existing Map or can instantiate its own Map. After any DynaBean processing has finished the Map can be retrieved and the DynaBean *facade* discarded.

If you need a new Map then to use....

DynaBean dynaBean = new LazyDynaMap(); // create DynaBean  
  
 dynaBean.set("foo", "bar"); // simple  
 dynaBean.set("customer", "title", "Mr"); // mapped  
 dynaBean.set("address", 0, addressLine1); // indexed  
  
 Map myMap = dynaBean.getMap() // retrieve the Map

*or* to use with an existing Map ....

Map myMap = .... // exisitng Map  
 DynaBean dynaBean = new LazyDynaMap(myMap); // wrap Map in DynaBean  
 dynaBean.set("foo", "bar"); // set properties

**3.** [**LazyDynaList**](http://docs.google.com/org/apache/commons/beanutils/LazyDynaList.html) is *lazy list* for [DynaBeans](http://docs.google.com/org/apache/commons/beanutils/DynaBean.html) java.util.Map's or POJO beans. See the [Javadoc](http://docs.google.com/LazyDynaList.html) for more details and example usage.

**4.** [**LazyDynaClass**](http://docs.google.com/org/apache/commons/beanutils/LazyDynaClass.html) extends [BasicDynaClass](http://docs.google.com/org/apache/commons/beanutils/BasicDynaClass.html) and implements the [MutableDynaClass](http://docs.google.com/MutableDynaClass.html) interface. It can be used with other DynaBean implementations, but it is the default DynaClass used by LazyDynaBean. When using the LazyDynaBean there may be no need to have anything to do with the DynaClass. However sometimes there is a requirement to set up the DynaClass first - perhaps to define the type of array for an indexed property, or if using the DynaBean in *restricted* mode (see note below) is required. Doing so is straight forward...

*Either* create a LazyDynaClass first...

MutableDynaClass dynaClass = new LazyDynaClass(); // create DynaClass  
  
 dynaClass.add("amount", java.lang.Integer.class); // add property  
 dynaClass.add("orders", OrderBean[].class); // add indexed property  
 dynaClass.add("orders", java.util.TreeMapp.class); // add mapped property  
  
 DynaBean dynaBean = new LazyDynaBean(dynaClass); // Create DynaBean with associated DynaClass

*or* create a LazyDynaBean and get the DynaClass...

DynaBean dynaBean = new LazyDynaBean(); // Create LazyDynaBean  
 MutableDynaClass dynaClass =  
 (MutableDynaClass)dynaBean.getDynaClass(); // get DynaClass  
  
 dynaClass.add("amount", java.lang.Integer.class); // add property  
 dynaClass.add("myBeans", myPackage.MyBean[].class); // add 'array' indexed property  
 dynaClass.add("myMap", java.util.TreeMapp.class); // add mapped property

**NOTE:** One feature of [MutableDynaClass](http://docs.google.com/org/apache/commons/beanutils/MutableDynaClass.html) is that it has a *Restricted* property. When the DynaClass is *restricted* no properties can be added or removed from the DynaClass. Neither the LazyDynaBean or LazyDynaMap will add properties automatically if the DynaClass is *restricted*.

# 4. Data Type Conversions

### 4.1 Background

So far, we've only considered the cases where the data types of the dynamically accessed properties are known, and where we can use Java casts to perform type conversions. What happens if you want to automatically perform type conversions when casting is not possible? The *BeanUtils* package provides a variety of APIs and design patterns for performing this task as well.

### 4.2 BeanUtils and ConvertUtils Conversions

A very common use case (and the situation that caused the initial creation of the *BeanUtils* package) was the desire to convert the set of request parameters that were included in a javax.servlet.HttpServletRequest received by a web application into a set of corresponding property setter calls on an arbitrary JavaBean. (This is one of the fundamental services provided by the [Struts Framework](http://struts.apache.org/), which uses *BeanUtils* internally to implement this functionality.)

In an HTTP request, the set of included parameters is made available as a series of String (or String array, if there is more than one value for the same parameter name) instances, which need to be converted to the underlying data type. The [BeanUtils](http://docs.google.com/org/apache/commons/beanutils/BeanUtils.html) class provides property setter methods that accept String values, and automatically convert them to appropriate property types for Java primitives (such as int or boolean), and property getter methods that perform the reverse conversion. Finally, a populate() method is provided that accepts a java.util.Map containing a set of property values (keyed by property name), and calls all of the appropriate setters whenever the underlying bean has a property with the same name as one of the request parameters. So, you can perform the all-in-one property setting operation like this:

HttpServletRequest request = ...;  
 MyBean bean = ...;  
 HashMap map = new HashMap();  
 Enumeration names = request.getParameterNames();  
 while (names.hasMoreElements()) {  
 String name = (String) names.nextElement();  
 map.put(name, request.getParameterValues(name));  
 }  
 BeanUtils.populate(bean, map);

The BeanUtils class relies on conversion methods defined in the [ConvertUtils](http://docs.google.com/org/apache/commons/beanutils/ConvertUtils.html) class to perform the actual conversions, and these methods are availablve for direct use as well. **WARNING** - It is likely that the hard coded use of ConvertUtils methods will be deprecated in the future, and replaced with a mechanism that allows you to plug in your own implementations of the [Converter](http://docs.google.com/org/apache/commons/beanutils/Converter.html) interface instead. Therefore, new code should not be written with reliance on ConvertUtils.

### 4.3 Defining Your Own Converters

The ConvertUtils class supports the ability to define and register your own String --> Object conversions for any given Java class. Once registered, such converters will be used transparently by all of the BeanUtils methods (including populate()). To create and register your own converter, follow these steps:

* Write a class that implements the [Converter](http://docs.google.com/org/apache/commons/beanutils/Converter.html) interface. The convert() method should accept the java.lang.Class object of your application class (i.e. the class that you want to convert to, and a String representing the incoming value to be converted.
* At application startup time, register an instance of your converter class by calling the ConvertUtils.register() method.

### 4.4 Locale Aware Conversions

The standard classes in org.apache.commons.beanutils are not locale aware. This gives them a cleaner interface and makes then easier to use in situations where the locale is not important.

Extended, locale-aware analogues can be found in [org.apache.commons.beanutils.locale](http://docs.google.com/locale/package-summary.html) . These are built along the same lines as the basic classes but support localization.

# 5. Utility Objects And Static Utility Classes

### Background

So far, the examples have covered the static utility classes (BeanUtils, ConvertUtils and PropertyUtils). These are easy to use but are somewhat inflexible. These all share the same registered converters and the same caches.

This functionality can also be accessed through utility objects (in fact, the static utility class use worker instances of these classes). For each static utility class, there is a corresponding class with the same functionality that can be instantiated:

|  |  |
| --- | --- |
| Static Utility Class | Utility Object |
| BeanUtils | BeanUtilsBean |
| ConvertUtils | ConvertUtilsBean |
| PropertyUtils | PropertyUtilsBean |

Creating an instances allow gives guarenteed control of the caching and registration to the code that creates it.

# 6. Collections

### 6.1 Comparing Beans

org.apache.commons.beanutils.BeanComparator is a Comparator implementation that compares beans based on a shared property value.

### 6.2 Operating On Collections Of Beans

The Closure interface in commons-collections encapsulates a block of code that executes on an arbitrary input Object. Commons-collections contains code that allows Closures to be applied to the contents of a Collection. For more details, see the [commons-collections](http://commons.apache.org/collections/) documentation.

BeanPropertyValueChangeClosure is a Closure that sets a specified property to a particular value. A typical usage is to combine this with commons-collections so that all the beans in a collection can have a particular property set to a particular value.

For example, set the activeEmployee property to TRUE for an entire collection:

// create the closure  
 BeanPropertyValueChangeClosure closure =  
 new BeanPropertyValueChangeClosure( "activeEmployee", Boolean.TRUE );  
  
 // update the Collection  
 CollectionUtils.forAllDo( peopleCollection, closure );

### 6.3 Querying Or Filtering Collections Of Beans

The Predicate interface in commons-collections encapsulates an evaluation of an input Object that returns either true or false. Commons-collections contains code that allows Predicates to be applied to be used to filter collections. For more details, see the [commons-collections](http://commons.apache.org/collections/) documentation.

BeanPropertyValueEqualsPredicate is a Predicate that evaluates a set property value against a given value. A typical usage is (in combination with commons-collections) to filter collections on the basis of a property value.

For example, to filter a collection to find all beans where active employee is false use:

BeanPropertyValueEqualsPredicate predicate =  
 new BeanPropertyValueEqualsPredicate( "activeEmployee", Boolean.FALSE );  
  
 // filter the Collection  
 CollectionUtils.filter( peopleCollection, predicate );

### 6.4 Transforming Collections Of Beans

The Transformer interface in commons-collections encapsulates the transformation of an input Object into an output object. Commons-collections contains code that allows Transformers to be applied produce a collection of outputs from a collection of inputs. For more details, see the [commons-collections](http://commons.apache.org/collections/) documentation.

BeanToPropertyTransformer is a Transformer implementation that transforms a bean into it's property value.

For example, to find all cities that are contained in the address of each person property of each bean in a collection:

// create the transformer  
 BeanToPropertyValueTransformer transformer = new BeanToPropertyValueTransformer( "person.address.city" );  
  
 // transform the Collection  
 Collection peoplesCities = CollectionUtils.collect( peopleCollection, transformer );

# 7. Frequently Asked Questions

### Why Can't BeanUtils Find My Method?

The *BeanUtils* package relies on *introspection* rather than *reflection*. This means that it will find only [*JavaBean* compliant](http://java.sun.com/products/javabeans) properties.

There are some subtleties of this specification that can catch out the unwary:

* A property can have only one set and one get method. Overloading is not allowed.
* The java.beans.Introspector searches widely for a custom *BeanInfo* class. If your class has the same name as another with a custom *BeanInfo* (typically a java API class) then the Introspector may use that instead of creating via reflection based on your class. If this happens, the only solution is to create your own *BeanInfo*.

### How Do I Set The BeanComparator Order To Be Ascending/Descending?

BeanComparator relies on an internal Comparator to perform the actual comparisions. By default, org.apache.commons.collections.comparators.ComparableComparator is used which imposes a natural order. If you want to change the order, then a custom Comparator should be created and passed into the appropriate constructor.

For example:

import org.apache.commons.collections.comparators.ComparableComparator;  
 import org.apache.commons.collections.comparators.ReverseComparator;  
 import org.apache.commons.beanutils.BeanComparator;  
 ...  
 BeanComparator reversedNaturalOrderBeanComparator  
 = new BeanComparator("propertyName", new ReverseComparator(new ComparableComparator()));  
 Collections.sort(myList, reversedNaturalOrderBeanComparator);  
 ...

[Skip navigation links](#1hmsyys)

* [Overview](http://docs.google.com/overview-summary.html)
* Package
* Class
* [Use](http://docs.google.com/package-use.html)
* [Tree](http://docs.google.com/package-tree.html)
* [Deprecated](http://docs.google.com/deprecated-list.html)
* [Index](http://docs.google.com/index-all.html)
* [Help](http://docs.google.com/help-doc.html)
* Prev Package
* [Next Package](http://docs.google.com/org/apache/commons/beanutils/converters/package-summary.html)
* [Frames](http://docs.google.com/index.html?org/apache/commons/beanutils/package-summary.html)
* [No Frames](http://docs.google.com/package-summary.html)
* [All Classes](http://docs.google.com/allclasses-noframe.html)

Copyright © 2000–2019 [The Apache Software Foundation](https://www.apache.org/). All rights reserved.