001/\*  
002 \* Licensed to the Apache Software Foundation (ASF) under one or more  
003 \* contributor license agreements. See the NOTICE file distributed with  
004 \* this work for additional information regarding copyright ownership.  
005 \* The ASF licenses this file to You under the Apache License, Version 2.0  
006 \* (the "License"); you may not use this file except in compliance with  
007 \* the License. You may obtain a copy of the License at  
008 \*  
009 \* http://www.apache.org/licenses/LICENSE-2.0  
010 \*  
011 \* Unless required by applicable law or agreed to in writing, software  
012 \* distributed under the License is distributed on an "AS IS" BASIS,  
013 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
014 \* See the License for the specific language governing permissions and  
015 \* limitations under the License.  
016 \*/  
017  
018package org.apache.commons.beanutils;  
019  
020  
021import java.beans.IndexedPropertyDescriptor;  
022import java.beans.IntrospectionException;  
023import java.beans.Introspector;  
024import java.beans.PropertyDescriptor;  
025import java.lang.reflect.Array;  
026import java.lang.reflect.InvocationTargetException;  
027import java.lang.reflect.Method;  
028import java.util.HashMap;  
029import java.util.Iterator;  
030import java.util.List;  
031import java.util.Map;  
032import java.util.Map.Entry;  
033import java.util.concurrent.CopyOnWriteArrayList;  
034  
035import org.apache.commons.beanutils.expression.DefaultResolver;  
036import org.apache.commons.beanutils.expression.Resolver;  
037import org.apache.commons.collections.FastHashMap;  
038import org.apache.commons.logging.Log;  
039import org.apache.commons.logging.LogFactory;  
040  
041  
042/\*\*  
043 \* Utility methods for using Java Reflection APIs to facilitate generic  
044 \* property getter and setter operations on Java objects. Much of this  
045 \* code was originally included in <code>BeanUtils</code>, but has been  
046 \* separated because of the volume of code involved.  
047 \* <p>  
048 \* In general, the objects that are examined and modified using these  
049 \* methods are expected to conform to the property getter and setter method  
050 \* naming conventions described in the JavaBeans Specification (Version 1.0.1).  
051 \* No data type conversions are performed, and there are no usage of any  
052 \* <code>PropertyEditor</code> classes that have been registered, although  
053 \* a convenient way to access the registered classes themselves is included.  
054 \* <p>  
055 \* For the purposes of this class, five formats for referencing a particular  
056 \* property value of a bean are defined, with the <i>default</i> layout of an  
057 \* identifying String in parentheses. However the notation for these formats  
058 \* and how they are resolved is now (since BeanUtils 1.8.0) controlled by  
059 \* the configured {@link Resolver} implementation:  
060 \* <ul>  
061 \* <li><strong>Simple (<code>name</code>)</strong> - The specified  
062 \* <code>name</code> identifies an individual property of a particular  
063 \* JavaBean. The name of the actual getter or setter method to be used  
064 \* is determined using standard JavaBeans instrospection, so that (unless  
065 \* overridden by a <code>BeanInfo</code> class, a property named "xyz"  
066 \* will have a getter method named <code>getXyz()</code> or (for boolean  
067 \* properties only) <code>isXyz()</code>, and a setter method named  
068 \* <code>setXyz()</code>.</li>  
069 \* <li><strong>Nested (<code>name1.name2.name3</code>)</strong> The first  
070 \* name element is used to select a property getter, as for simple  
071 \* references above. The object returned for this property is then  
072 \* consulted, using the same approach, for a property getter for a  
073 \* property named <code>name2</code>, and so on. The property value that  
074 \* is ultimately retrieved or modified is the one identified by the  
075 \* last name element.</li>  
076 \* <li><strong>Indexed (<code>name[index]</code>)</strong> - The underlying  
077 \* property value is assumed to be an array, or this JavaBean is assumed  
078 \* to have indexed property getter and setter methods. The appropriate  
079 \* (zero-relative) entry in the array is selected. <code>List</code>  
080 \* objects are now also supported for read/write. You simply need to define  
081 \* a getter that returns the <code>List</code></li>  
082 \* <li><strong>Mapped (<code>name(key)</code>)</strong> - The JavaBean  
083 \* is assumed to have an property getter and setter methods with an  
084 \* additional attribute of type <code>java.lang.String</code>.</li>  
085 \* <li><strong>Combined (<code>name1.name2[index].name3(key)</code>)</strong> -  
086 \* Combining mapped, nested, and indexed references is also  
087 \* supported.</li>  
088 \* </ul>  
089 \*  
090 \* @version $Id$  
091 \* @see Resolver  
092 \* @see PropertyUtils  
093 \* @since 1.7  
094 \*/  
095  
096public class PropertyUtilsBean {  
097  
098 private Resolver resolver = new DefaultResolver();  
099  
100 // --------------------------------------------------------- Class Methods  
101  
102 /\*\*  
103 \* Return the PropertyUtils bean instance.  
104 \* @return The PropertyUtils bean instance  
105 \*/  
106 protected static PropertyUtilsBean getInstance() {  
107 return BeanUtilsBean.getInstance().getPropertyUtils();  
108 }  
109  
110 // --------------------------------------------------------- Variables  
111  
112 /\*\*  
113 \* The cache of PropertyDescriptor arrays for beans we have already  
114 \* introspected, keyed by the java.lang.Class of this object.  
115 \*/  
116 private WeakFastHashMap<Class<?>, BeanIntrospectionData> descriptorsCache = null;  
117 private WeakFastHashMap<Class<?>, FastHashMap> mappedDescriptorsCache = null;  
118  
119 /\*\* An empty object array \*/  
120 private static final Object[] EMPTY\_OBJECT\_ARRAY = new Object[0];  
121  
122 /\*\* Log instance \*/  
123 private final Log log = LogFactory.getLog(PropertyUtils.class);  
124  
125 /\*\* The list with BeanIntrospector objects. \*/  
126 private final List<BeanIntrospector> introspectors;  
127  
128 // ---------------------------------------------------------- Constructors  
129  
130 /\*\* Base constructor \*/  
131 public PropertyUtilsBean() {  
132 descriptorsCache = new WeakFastHashMap<Class<?>, BeanIntrospectionData>();  
133 descriptorsCache.setFast(true);  
134 mappedDescriptorsCache = new WeakFastHashMap<Class<?>, FastHashMap>();  
135 mappedDescriptorsCache.setFast(true);  
136 introspectors = new CopyOnWriteArrayList<BeanIntrospector>();  
137 resetBeanIntrospectors();  
138 }  
139  
140  
141 // --------------------------------------------------------- Public Methods  
142  
143  
144 /\*\*  
145 \* Return the configured {@link Resolver} implementation used by BeanUtils.  
146 \* <p>  
147 \* The {@link Resolver} handles the <i>property name</i>  
148 \* expressions and the implementation in use effectively  
149 \* controls the dialect of the <i>expression language</i>  
150 \* that BeanUtils recongnises.  
151 \* <p>  
152 \* {@link DefaultResolver} is the default implementation used.  
153 \*  
154 \* @return resolver The property expression resolver.  
155 \* @since 1.8.0  
156 \*/  
157 public Resolver getResolver() {  
158 return resolver;  
159 }  
160  
161 /\*\*  
162 \* Configure the {@link Resolver} implementation used by BeanUtils.  
163 \* <p>  
164 \* The {@link Resolver} handles the <i>property name</i>  
165 \* expressions and the implementation in use effectively  
166 \* controls the dialect of the <i>expression language</i>  
167 \* that BeanUtils recongnises.  
168 \* <p>  
169 \* {@link DefaultResolver} is the default implementation used.  
170 \*  
171 \* @param resolver The property expression resolver.  
172 \* @since 1.8.0  
173 \*/  
174 public void setResolver(final Resolver resolver) {  
175 if (resolver == null) {  
176 this.resolver = new DefaultResolver();  
177 } else {  
178 this.resolver = resolver;  
179 }  
180 }  
181  
182 /\*\*  
183 \* Resets the {@link BeanIntrospector} objects registered at this instance. After this  
184 \* method was called, only the default {@code BeanIntrospector} is registered.  
185 \*  
186 \* @since 1.9  
187 \*/  
188 public final void resetBeanIntrospectors() {  
189 introspectors.clear();  
190 introspectors.add(DefaultBeanIntrospector.INSTANCE);  
191 introspectors.add(SuppressPropertiesBeanIntrospector.SUPPRESS\_CLASS);  
192 }  
193  
194 /\*\*  
195 \* Adds a <code>BeanIntrospector</code>. This object is invoked when the  
196 \* property descriptors of a class need to be obtained.  
197 \*  
198 \* @param introspector the <code>BeanIntrospector</code> to be added (must  
199 \* not be <b>null</b>  
200 \* @throws IllegalArgumentException if the argument is <b>null</b>  
201 \* @since 1.9  
202 \*/  
203 public void addBeanIntrospector(final BeanIntrospector introspector) {  
204 if (introspector == null) {  
205 throw new IllegalArgumentException(  
206 "BeanIntrospector must not be null!");  
207 }  
208 introspectors.add(introspector);  
209 }  
210  
211 /\*\*  
212 \* Removes the specified <code>BeanIntrospector</code>.  
213 \*  
214 \* @param introspector the <code>BeanIntrospector</code> to be removed  
215 \* @return <b>true</b> if the <code>BeanIntrospector</code> existed and  
216 \* could be removed, <b>false</b> otherwise  
217 \* @since 1.9  
218 \*/  
219 public boolean removeBeanIntrospector(final BeanIntrospector introspector) {  
220 return introspectors.remove(introspector);  
221 }  
222  
223 /\*\*  
224 \* Clear any cached property descriptors information for all classes  
225 \* loaded by any class loaders. This is useful in cases where class  
226 \* loaders are thrown away to implement class reloading.  
227 \*/  
228 public void clearDescriptors() {  
229  
230 descriptorsCache.clear();  
231 mappedDescriptorsCache.clear();  
232 Introspector.flushCaches();  
233  
234 }  
235  
236  
237 /\*\*  
238 \* <p>Copy property values from the "origin" bean to the "destination" bean  
239 \* for all cases where the property names are the same (even though the  
240 \* actual getter and setter methods might have been customized via  
241 \* <code>BeanInfo</code> classes). No conversions are performed on the  
242 \* actual property values -- it is assumed that the values retrieved from  
243 \* the origin bean are assignment-compatible with the types expected by  
244 \* the destination bean.</p>  
245 \*  
246 \* <p>If the origin "bean" is actually a <code>Map</code>, it is assumed  
247 \* to contain String-valued <strong>simple</strong> property names as the keys, pointing  
248 \* at the corresponding property values that will be set in the destination  
249 \* bean.<strong>Note</strong> that this method is intended to perform  
250 \* a "shallow copy" of the properties and so complex properties  
251 \* (for example, nested ones) will not be copied.</p>  
252 \*  
253 \* <p>Note, that this method will not copy a List to a List, or an Object[]  
254 \* to an Object[]. It's specifically for copying JavaBean properties. </p>  
255 \*  
256 \* @param dest Destination bean whose properties are modified  
257 \* @param orig Origin bean whose properties are retrieved  
258 \*  
259 \* @throws IllegalAccessException if the caller does not have  
260 \* access to the property accessor method  
261 \* @throws IllegalArgumentException if the <code>dest</code> or  
262 \* <code>orig</code> argument is null  
263 \* @throws InvocationTargetException if the property accessor method  
264 \* throws an exception  
265 \* @throws NoSuchMethodException if an accessor method for this  
266 \* propety cannot be found  
267 \*/  
268 public void copyProperties(final Object dest, final Object orig)  
269 throws IllegalAccessException, InvocationTargetException,  
270 NoSuchMethodException {  
271  
272 if (dest == null) {  
273 throw new IllegalArgumentException  
274 ("No destination bean specified");  
275 }  
276 if (orig == null) {  
277 throw new IllegalArgumentException("No origin bean specified");  
278 }  
279  
280 if (orig instanceof DynaBean) {  
281 final DynaProperty[] origDescriptors =  
282 ((DynaBean) orig).getDynaClass().getDynaProperties();  
283 for (DynaProperty origDescriptor : origDescriptors) {  
284 final String name = origDescriptor.getName();  
285 if (isReadable(orig, name) && isWriteable(dest, name)) {  
286 try {  
287 final Object value = ((DynaBean) orig).get(name);  
288 if (dest instanceof DynaBean) {  
289 ((DynaBean) dest).set(name, value);  
290 } else {  
291 setSimpleProperty(dest, name, value);  
292 }  
293 } catch (final NoSuchMethodException e) {  
294 if (log.isDebugEnabled()) {  
295 log.debug("Error writing to '" + name + "' on class '" + dest.getClass() + "'", e);  
296 }  
297 }  
298 }  
299 }  
300 } else if (orig instanceof Map) {  
301 final Iterator<?> entries = ((Map<?, ?>) orig).entrySet().iterator();  
302 while (entries.hasNext()) {  
303 final Map.Entry<?, ?> entry = (Entry<?, ?>) entries.next();  
304 final String name = (String)entry.getKey();  
305 if (isWriteable(dest, name)) {  
306 try {  
307 if (dest instanceof DynaBean) {  
308 ((DynaBean) dest).set(name, entry.getValue());  
309 } else {  
310 setSimpleProperty(dest, name, entry.getValue());  
311 }  
312 } catch (final NoSuchMethodException e) {  
313 if (log.isDebugEnabled()) {  
314 log.debug("Error writing to '" + name + "' on class '" + dest.getClass() + "'", e);  
315 }  
316 }  
317 }  
318 }  
319 } else /\* if (orig is a standard JavaBean) \*/ {  
320 final PropertyDescriptor[] origDescriptors =  
321 getPropertyDescriptors(orig);  
322 for (PropertyDescriptor origDescriptor : origDescriptors) {  
323 final String name = origDescriptor.getName();  
324 if (isReadable(orig, name) && isWriteable(dest, name)) {  
325 try {  
326 final Object value = getSimpleProperty(orig, name);  
327 if (dest instanceof DynaBean) {  
328 ((DynaBean) dest).set(name, value);  
329 } else {  
330 setSimpleProperty(dest, name, value);  
331 }  
332 } catch (final NoSuchMethodException e) {  
333 if (log.isDebugEnabled()) {  
334 log.debug("Error writing to '" + name + "' on class '" + dest.getClass() + "'", e);  
335 }  
336 }  
337 }  
338 }  
339 }  
340  
341 }  
342  
343  
344 /\*\*  
345 \* <p>Return the entire set of properties for which the specified bean  
346 \* provides a read method. This map contains the unconverted property  
347 \* values for all properties for which a read method is provided  
348 \* (i.e. where the <code>getReadMethod()</code> returns non-null).</p>  
349 \*  
350 \* <p><strong>FIXME</strong> - Does not account for mapped properties.</p>  
351 \*  
352 \* @param bean Bean whose properties are to be extracted  
353 \* @return The set of properties for the bean  
354 \*  
355 \* @throws IllegalAccessException if the caller does not have  
356 \* access to the property accessor method  
357 \* @throws IllegalArgumentException if <code>bean</code> is null  
358 \* @throws InvocationTargetException if the property accessor method  
359 \* throws an exception  
360 \* @throws NoSuchMethodException if an accessor method for this  
361 \* propety cannot be found  
362 \*/  
363 public Map<String, Object> describe(final Object bean)  
364 throws IllegalAccessException, InvocationTargetException,  
365 NoSuchMethodException {  
366  
367 if (bean == null) {  
368 throw new IllegalArgumentException("No bean specified");  
369 }  
370 final Map<String, Object> description = new HashMap<String, Object>();  
371 if (bean instanceof DynaBean) {  
372 final DynaProperty[] descriptors =  
373 ((DynaBean) bean).getDynaClass().getDynaProperties();  
374 for (DynaProperty descriptor : descriptors) {  
375 final String name = descriptor.getName();  
376 description.put(name, getProperty(bean, name));  
377 }  
378 } else {  
379 final PropertyDescriptor[] descriptors =  
380 getPropertyDescriptors(bean);  
381 for (PropertyDescriptor descriptor : descriptors) {  
382 final String name = descriptor.getName();  
383 if (descriptor.getReadMethod() != null) {  
384 description.put(name, getProperty(bean, name));  
385 }  
386 }  
387 }  
388 return (description);  
389  
390 }  
391  
392  
393 /\*\*  
394 \* Return the value of the specified indexed property of the specified  
395 \* bean, with no type conversions. The zero-relative index of the  
396 \* required value must be included (in square brackets) as a suffix to  
397 \* the property name, or <code>IllegalArgumentException</code> will be  
398 \* thrown. In addition to supporting the JavaBeans specification, this  
399 \* method has been extended to support <code>List</code> objects as well.  
400 \*  
401 \* @param bean Bean whose property is to be extracted  
402 \* @param name <code>propertyname[index]</code> of the property value  
403 \* to be extracted  
404 \* @return the indexed property value  
405 \*  
406 \* @throws IndexOutOfBoundsException if the specified index  
407 \* is outside the valid range for the underlying array or List  
408 \* @throws IllegalAccessException if the caller does not have  
409 \* access to the property accessor method  
410 \* @throws IllegalArgumentException if <code>bean</code> or  
411 \* <code>name</code> is null  
412 \* @throws InvocationTargetException if the property accessor method  
413 \* throws an exception  
414 \* @throws NoSuchMethodException if an accessor method for this  
415 \* propety cannot be found  
416 \*/  
417 public Object getIndexedProperty(final Object bean, String name)  
418 throws IllegalAccessException, InvocationTargetException,  
419 NoSuchMethodException {  
420  
421 if (bean == null) {  
422 throw new IllegalArgumentException("No bean specified");  
423 }  
424 if (name == null) {  
425 throw new IllegalArgumentException("No name specified for bean class '" +  
426 bean.getClass() + "'");  
427 }  
428  
429 // Identify the index of the requested individual property  
430 int index = -1;  
431 try {  
432 index = resolver.getIndex(name);  
433 } catch (final IllegalArgumentException e) {  
434 throw new IllegalArgumentException("Invalid indexed property '" +  
435 name + "' on bean class '" + bean.getClass() + "' " +  
436 e.getMessage());  
437 }  
438 if (index < 0) {  
439 throw new IllegalArgumentException("Invalid indexed property '" +  
440 name + "' on bean class '" + bean.getClass() + "'");  
441 }  
442  
443 // Isolate the name  
444 name = resolver.getProperty(name);  
445  
446 // Request the specified indexed property value  
447 return (getIndexedProperty(bean, name, index));  
448  
449 }  
450  
451  
452 /\*\*  
453 \* Return the value of the specified indexed property of the specified  
454 \* bean, with no type conversions. In addition to supporting the JavaBeans  
455 \* specification, this method has been extended to support  
456 \* <code>List</code> objects as well.  
457 \*  
458 \* @param bean Bean whose property is to be extracted  
459 \* @param name Simple property name of the property value to be extracted  
460 \* @param index Index of the property value to be extracted  
461 \* @return the indexed property value  
462 \*  
463 \* @throws IndexOutOfBoundsException if the specified index  
464 \* is outside the valid range for the underlying property  
465 \* @throws IllegalAccessException if the caller does not have  
466 \* access to the property accessor method  
467 \* @throws IllegalArgumentException if <code>bean</code> or  
468 \* <code>name</code> is null  
469 \* @throws InvocationTargetException if the property accessor method  
470 \* throws an exception  
471 \* @throws NoSuchMethodException if an accessor method for this  
472 \* propety cannot be found  
473 \*/  
474 public Object getIndexedProperty(final Object bean,  
475 final String name, final int index)  
476 throws IllegalAccessException, InvocationTargetException,  
477 NoSuchMethodException {  
478  
479 if (bean == null) {  
480 throw new IllegalArgumentException("No bean specified");  
481 }  
482 if (name == null || name.length() == 0) {  
483 if (bean.getClass().isArray()) {  
484 return Array.get(bean, index);  
485 } else if (bean instanceof List) {  
486 return ((List<?>)bean).get(index);  
487 }  
488 }  
489 if (name == null) {  
490 throw new IllegalArgumentException("No name specified for bean class '" +  
491 bean.getClass() + "'");  
492 }  
493  
494 // Handle DynaBean instances specially  
495 if (bean instanceof DynaBean) {  
496 final DynaProperty descriptor =  
497 ((DynaBean) bean).getDynaClass().getDynaProperty(name);  
498 if (descriptor == null) {  
499 throw new NoSuchMethodException("Unknown property '" +  
500 name + "' on bean class '" + bean.getClass() + "'");  
501 }  
502 return (((DynaBean) bean).get(name, index));  
503 }  
504  
505 // Retrieve the property descriptor for the specified property  
506 final PropertyDescriptor descriptor =  
507 getPropertyDescriptor(bean, name);  
508 if (descriptor == null) {  
509 throw new NoSuchMethodException("Unknown property '" +  
510 name + "' on bean class '" + bean.getClass() + "'");  
511 }  
512  
513 // Call the indexed getter method if there is one  
514 if (descriptor instanceof IndexedPropertyDescriptor) {  
515 Method readMethod = ((IndexedPropertyDescriptor) descriptor).  
516 getIndexedReadMethod();  
517 readMethod = MethodUtils.getAccessibleMethod(bean.getClass(), readMethod);  
518 if (readMethod != null) {  
519 final Object[] subscript = new Object[1];  
520 subscript[0] = new Integer(index);  
521 try {  
522 return (invokeMethod(readMethod,bean, subscript));  
523 } catch (final InvocationTargetException e) {  
524 if (e.getTargetException() instanceof  
525 IndexOutOfBoundsException) {  
526 throw (IndexOutOfBoundsException)  
527 e.getTargetException();  
528 } else {  
529 throw e;  
530 }  
531 }  
532 }  
533 }  
534  
535 // Otherwise, the underlying property must be an array  
536 final Method readMethod = getReadMethod(bean.getClass(), descriptor);  
537 if (readMethod == null) {  
538 throw new NoSuchMethodException("Property '" + name + "' has no " +  
539 "getter method on bean class '" + bean.getClass() + "'");  
540 }  
541  
542 // Call the property getter and return the value  
543 final Object value = invokeMethod(readMethod, bean, EMPTY\_OBJECT\_ARRAY);  
544 if (!value.getClass().isArray()) {  
545 if (!(value instanceof java.util.List)) {  
546 throw new IllegalArgumentException("Property '" + name +  
547 "' is not indexed on bean class '" + bean.getClass() + "'");  
548 } else {  
549 //get the List's value  
550 return ((java.util.List<?>) value).get(index);  
551 }  
552 } else {  
553 //get the array's value  
554 try {  
555 return (Array.get(value, index));  
556 } catch (final ArrayIndexOutOfBoundsException e) {  
557 throw new ArrayIndexOutOfBoundsException("Index: " +  
558 index + ", Size: " + Array.getLength(value) +  
559 " for property '" + name + "'");  
560 }  
561 }  
562  
563 }  
564  
565  
566 /\*\*  
567 \* Return the value of the specified mapped property of the  
568 \* specified bean, with no type conversions. The key of the  
569 \* required value must be included (in brackets) as a suffix to  
570 \* the property name, or <code>IllegalArgumentException</code> will be  
571 \* thrown.  
572 \*  
573 \* @param bean Bean whose property is to be extracted  
574 \* @param name <code>propertyname(key)</code> of the property value  
575 \* to be extracted  
576 \* @return the mapped property value  
577 \*  
578 \* @throws IllegalAccessException if the caller does not have  
579 \* access to the property accessor method  
580 \* @throws InvocationTargetException if the property accessor method  
581 \* throws an exception  
582 \* @throws NoSuchMethodException if an accessor method for this  
583 \* propety cannot be found  
584 \*/  
585 public Object getMappedProperty(final Object bean, String name)  
586 throws IllegalAccessException, InvocationTargetException,  
587 NoSuchMethodException {  
588  
589 if (bean == null) {  
590 throw new IllegalArgumentException("No bean specified");  
591 }  
592 if (name == null) {  
593 throw new IllegalArgumentException("No name specified for bean class '" +  
594 bean.getClass() + "'");  
595 }  
596  
597 // Identify the key of the requested individual property  
598 String key = null;  
599 try {  
600 key = resolver.getKey(name);  
601 } catch (final IllegalArgumentException e) {  
602 throw new IllegalArgumentException  
603 ("Invalid mapped property '" + name +  
604 "' on bean class '" + bean.getClass() + "' " + e.getMessage());  
605 }  
606 if (key == null) {  
607 throw new IllegalArgumentException("Invalid mapped property '" +  
608 name + "' on bean class '" + bean.getClass() + "'");  
609 }  
610  
611 // Isolate the name  
612 name = resolver.getProperty(name);  
613  
614 // Request the specified indexed property value  
615 return (getMappedProperty(bean, name, key));  
616  
617 }  
618  
619  
620 /\*\*  
621 \* Return the value of the specified mapped property of the specified  
622 \* bean, with no type conversions.  
623 \*  
624 \* @param bean Bean whose property is to be extracted  
625 \* @param name Mapped property name of the property value to be extracted  
626 \* @param key Key of the property value to be extracted  
627 \* @return the mapped property value  
628 \*  
629 \* @throws IllegalAccessException if the caller does not have  
630 \* access to the property accessor method  
631 \* @throws InvocationTargetException if the property accessor method  
632 \* throws an exception  
633 \* @throws NoSuchMethodException if an accessor method for this  
634 \* propety cannot be found  
635 \*/  
636 public Object getMappedProperty(final Object bean,  
637 final String name, final String key)  
638 throws IllegalAccessException, InvocationTargetException,  
639 NoSuchMethodException {  
640  
641 if (bean == null) {  
642 throw new IllegalArgumentException("No bean specified");  
643 }  
644 if (name == null) {  
645 throw new IllegalArgumentException("No name specified for bean class '" +  
646 bean.getClass() + "'");  
647 }  
648 if (key == null) {  
649 throw new IllegalArgumentException("No key specified for property '" +  
650 name + "' on bean class " + bean.getClass() + "'");  
651 }  
652  
653 // Handle DynaBean instances specially  
654 if (bean instanceof DynaBean) {  
655 final DynaProperty descriptor =  
656 ((DynaBean) bean).getDynaClass().getDynaProperty(name);  
657 if (descriptor == null) {  
658 throw new NoSuchMethodException("Unknown property '" +  
659 name + "'+ on bean class '" + bean.getClass() + "'");  
660 }  
661 return (((DynaBean) bean).get(name, key));  
662 }  
663  
664 Object result = null;  
665  
666 // Retrieve the property descriptor for the specified property  
667 final PropertyDescriptor descriptor = getPropertyDescriptor(bean, name);  
668 if (descriptor == null) {  
669 throw new NoSuchMethodException("Unknown property '" +  
670 name + "'+ on bean class '" + bean.getClass() + "'");  
671 }  
672  
673 if (descriptor instanceof MappedPropertyDescriptor) {  
674 // Call the keyed getter method if there is one  
675 Method readMethod = ((MappedPropertyDescriptor) descriptor).  
676 getMappedReadMethod();  
677 readMethod = MethodUtils.getAccessibleMethod(bean.getClass(), readMethod);  
678 if (readMethod != null) {  
679 final Object[] keyArray = new Object[1];  
680 keyArray[0] = key;  
681 result = invokeMethod(readMethod, bean, keyArray);  
682 } else {  
683 throw new NoSuchMethodException("Property '" + name +  
684 "' has no mapped getter method on bean class '" +  
685 bean.getClass() + "'");  
686 }  
687 } else {  
688 /\* means that the result has to be retrieved from a map \*/  
689 final Method readMethod = getReadMethod(bean.getClass(), descriptor);  
690 if (readMethod != null) {  
691 final Object invokeResult = invokeMethod(readMethod, bean, EMPTY\_OBJECT\_ARRAY);  
692 /\* test and fetch from the map \*/  
693 if (invokeResult instanceof java.util.Map) {  
694 result = ((java.util.Map<?, ?>)invokeResult).get(key);  
695 }  
696 } else {  
697 throw new NoSuchMethodException("Property '" + name +  
698 "' has no mapped getter method on bean class '" +  
699 bean.getClass() + "'");  
700 }  
701 }  
702 return result;  
703  
704 }  
705  
706  
707 /\*\*  
708 \* <p>Return the mapped property descriptors for this bean class.</p>  
709 \*  
710 \* <p><strong>FIXME</strong> - Does not work with DynaBeans.</p>  
711 \*  
712 \* @param beanClass Bean class to be introspected  
713 \* @return the mapped property descriptors  
714 \* @deprecated This method should not be exposed  
715 \*/  
716 @Deprecated  
717 public FastHashMap getMappedPropertyDescriptors(final Class<?> beanClass) {  
718  
719 if (beanClass == null) {  
720 return null;  
721 }  
722  
723 // Look up any cached descriptors for this bean class  
724 return mappedDescriptorsCache.get(beanClass);  
725  
726 }  
727  
728  
729 /\*\*  
730 \* <p>Return the mapped property descriptors for this bean.</p>  
731 \*  
732 \* <p><strong>FIXME</strong> - Does not work with DynaBeans.</p>  
733 \*  
734 \* @param bean Bean to be introspected  
735 \* @return the mapped property descriptors  
736 \* @deprecated This method should not be exposed  
737 \*/  
738 @Deprecated  
739 public FastHashMap getMappedPropertyDescriptors(final Object bean) {  
740  
741 if (bean == null) {  
742 return null;  
743 }  
744 return (getMappedPropertyDescriptors(bean.getClass()));  
745  
746 }  
747  
748  
749 /\*\*  
750 \* Return the value of the (possibly nested) property of the specified  
751 \* name, for the specified bean, with no type conversions.  
752 \*  
753 \* @param bean Bean whose property is to be extracted  
754 \* @param name Possibly nested name of the property to be extracted  
755 \* @return the nested property value  
756 \*  
757 \* @throws IllegalAccessException if the caller does not have  
758 \* access to the property accessor method  
759 \* @throws IllegalArgumentException if <code>bean</code> or  
760 \* <code>name</code> is null  
761 \* @throws NestedNullException if a nested reference to a  
762 \* property returns null  
763 \* @throws InvocationTargetException  
764 \* if the property accessor method throws an exception  
765 \* @throws NoSuchMethodException if an accessor method for this  
766 \* propety cannot be found  
767 \*/  
768 public Object getNestedProperty(Object bean, String name)  
769 throws IllegalAccessException, InvocationTargetException,  
770 NoSuchMethodException {  
771  
772 if (bean == null) {  
773 throw new IllegalArgumentException("No bean specified");  
774 }  
775 if (name == null) {  
776 throw new IllegalArgumentException("No name specified for bean class '" +  
777 bean.getClass() + "'");  
778 }  
779  
780 // Resolve nested references  
781 while (resolver.hasNested(name)) {  
782 final String next = resolver.next(name);  
783 Object nestedBean = null;  
784 if (bean instanceof Map) {  
785 nestedBean = getPropertyOfMapBean((Map<?, ?>) bean, next);  
786 } else if (resolver.isMapped(next)) {  
787 nestedBean = getMappedProperty(bean, next);  
788 } else if (resolver.isIndexed(next)) {  
789 nestedBean = getIndexedProperty(bean, next);  
790 } else {  
791 nestedBean = getSimpleProperty(bean, next);  
792 }  
793 if (nestedBean == null) {  
794 throw new NestedNullException  
795 ("Null property value for '" + name +  
796 "' on bean class '" + bean.getClass() + "'");  
797 }  
798 bean = nestedBean;  
799 name = resolver.remove(name);  
800 }  
801  
802 if (bean instanceof Map) {  
803 bean = getPropertyOfMapBean((Map<?, ?>) bean, name);  
804 } else if (resolver.isMapped(name)) {  
805 bean = getMappedProperty(bean, name);  
806 } else if (resolver.isIndexed(name)) {  
807 bean = getIndexedProperty(bean, name);  
808 } else {  
809 bean = getSimpleProperty(bean, name);  
810 }  
811 return bean;  
812  
813 }  
814  
815 /\*\*  
816 \* This method is called by getNestedProperty and setNestedProperty to  
817 \* define what it means to get a property from an object which implements  
818 \* Map. See setPropertyOfMapBean for more information.  
819 \*  
820 \* @param bean Map bean  
821 \* @param propertyName The property name  
822 \* @return the property value  
823 \*  
824 \* @throws IllegalArgumentException when the propertyName is regarded as  
825 \* being invalid.  
826 \*  
827 \* @throws IllegalAccessException just in case subclasses override this  
828 \* method to try to access real getter methods and find permission is denied.  
829 \*  
830 \* @throws InvocationTargetException just in case subclasses override this  
831 \* method to try to access real getter methods, and find it throws an  
832 \* exception when invoked.  
833 \*  
834 \* @throws NoSuchMethodException just in case subclasses override this  
835 \* method to try to access real getter methods, and want to fail if  
836 \* no simple method is available.  
837 \* @since 1.8.0  
838 \*/  
839 protected Object getPropertyOfMapBean(final Map<?, ?> bean, String propertyName)  
840 throws IllegalArgumentException, IllegalAccessException,  
841 InvocationTargetException, NoSuchMethodException {  
842  
843 if (resolver.isMapped(propertyName)) {  
844 final String name = resolver.getProperty(propertyName);  
845 if (name == null || name.length() == 0) {  
846 propertyName = resolver.getKey(propertyName);  
847 }  
848 }  
849  
850 if (resolver.isIndexed(propertyName) ||  
851 resolver.isMapped(propertyName)) {  
852 throw new IllegalArgumentException(  
853 "Indexed or mapped properties are not supported on"  
854 + " objects of type Map: " + propertyName);  
855 }  
856  
857 return bean.get(propertyName);  
858 }  
859  
860  
861  
862 /\*\*  
863 \* Return the value of the specified property of the specified bean,  
864 \* no matter which property reference format is used, with no  
865 \* type conversions.  
866 \*  
867 \* @param bean Bean whose property is to be extracted  
868 \* @param name Possibly indexed and/or nested name of the property  
869 \* to be extracted  
870 \* @return the property value  
871 \*  
872 \* @throws IllegalAccessException if the caller does not have  
873 \* access to the property accessor method  
874 \* @throws IllegalArgumentException if <code>bean</code> or  
875 \* <code>name</code> is null  
876 \* @throws InvocationTargetException if the property accessor method  
877 \* throws an exception  
878 \* @throws NoSuchMethodException if an accessor method for this  
879 \* propety cannot be found  
880 \*/  
881 public Object getProperty(final Object bean, final String name)  
882 throws IllegalAccessException, InvocationTargetException,  
883 NoSuchMethodException {  
884  
885 return (getNestedProperty(bean, name));  
886  
887 }  
888  
889  
890 /\*\*  
891 \* <p>Retrieve the property descriptor for the specified property of the  
892 \* specified bean, or return <code>null</code> if there is no such  
893 \* descriptor. This method resolves indexed and nested property  
894 \* references in the same manner as other methods in this class, except  
895 \* that if the last (or only) name element is indexed, the descriptor  
896 \* for the last resolved property itself is returned.</p>  
897 \*  
898 \* <p><strong>FIXME</strong> - Does not work with DynaBeans.</p>  
899 \*  
900 \* <p>Note that for Java 8 and above, this method no longer return  
901 \* IndexedPropertyDescriptor for {@link List}-typed properties, only for  
902 \* properties typed as native array. (BEANUTILS-492).  
903 \*  
904 \* @param bean Bean for which a property descriptor is requested  
905 \* @param name Possibly indexed and/or nested name of the property for  
906 \* which a property descriptor is requested  
907 \* @return the property descriptor  
908 \*  
909 \* @throws IllegalAccessException if the caller does not have  
910 \* access to the property accessor method  
911 \* @throws IllegalArgumentException if <code>bean</code> or  
912 \* <code>name</code> is null  
913 \* @throws IllegalArgumentException if a nested reference to a  
914 \* property returns null  
915 \* @throws InvocationTargetException if the property accessor method  
916 \* throws an exception  
917 \* @throws NoSuchMethodException if an accessor method for this  
918 \* propety cannot be found  
919 \*/  
920 public PropertyDescriptor getPropertyDescriptor(Object bean,  
921 String name)  
922 throws IllegalAccessException, InvocationTargetException,  
923 NoSuchMethodException {  
924  
925 if (bean == null) {  
926 throw new IllegalArgumentException("No bean specified");  
927 }  
928 if (name == null) {  
929 throw new IllegalArgumentException("No name specified for bean class '" +  
930 bean.getClass() + "'");  
931 }  
932  
933 // Resolve nested references  
934 while (resolver.hasNested(name)) {  
935 final String next = resolver.next(name);  
936 final Object nestedBean = getProperty(bean, next);  
937 if (nestedBean == null) {  
938 throw new NestedNullException  
939 ("Null property value for '" + next +  
940 "' on bean class '" + bean.getClass() + "'");  
941 }  
942 bean = nestedBean;  
943 name = resolver.remove(name);  
944 }  
945  
946 // Remove any subscript from the final name value  
947 name = resolver.getProperty(name);  
948  
949 // Look up and return this property from our cache  
950 // creating and adding it to the cache if not found.  
951 if (name == null) {  
952 return (null);  
953 }  
954  
955 final BeanIntrospectionData data = getIntrospectionData(bean.getClass());  
956 PropertyDescriptor result = data.getDescriptor(name);  
957 if (result != null) {  
958 return result;  
959 }  
960  
961 FastHashMap mappedDescriptors =  
962 getMappedPropertyDescriptors(bean);  
963 if (mappedDescriptors == null) {  
964 mappedDescriptors = new FastHashMap();  
965 mappedDescriptors.setFast(true);  
966 mappedDescriptorsCache.put(bean.getClass(), mappedDescriptors);  
967 }  
968 result = (PropertyDescriptor) mappedDescriptors.get(name);  
969 if (result == null) {  
970 // not found, try to create it  
971 try {  
972 result = new MappedPropertyDescriptor(name, bean.getClass());  
973 } catch (final IntrospectionException ie) {  
974 /\* Swallow IntrospectionException  
975 \* TODO: Why?  
976 \*/  
977 }  
978 if (result != null) {  
979 mappedDescriptors.put(name, result);  
980 }  
981 }  
982  
983 return result;  
984  
985 }  
986  
987  
988 /\*\*  
989 \* <p>Retrieve the property descriptors for the specified class,  
990 \* introspecting and caching them the first time a particular bean class  
991 \* is encountered.</p>  
992 \*  
993 \* <p><strong>FIXME</strong> - Does not work with DynaBeans.</p>  
994 \*  
995 \* @param beanClass Bean class for which property descriptors are requested  
996 \* @return the property descriptors  
997 \*  
998 \* @throws IllegalArgumentException if <code>beanClass</code> is null  
999 \*/  
1000 public PropertyDescriptor[]  
1001 getPropertyDescriptors(final Class<?> beanClass) {  
1002  
1003 return getIntrospectionData(beanClass).getDescriptors();  
1004  
1005 }  
1006  
1007 /\*\*  
1008 \* <p>Retrieve the property descriptors for the specified bean,  
1009 \* introspecting and caching them the first time a particular bean class  
1010 \* is encountered.</p>  
1011 \*  
1012 \* <p><strong>FIXME</strong> - Does not work with DynaBeans.</p>  
1013 \*  
1014 \* @param bean Bean for which property descriptors are requested  
1015 \* @return the property descriptors  
1016 \*  
1017 \* @throws IllegalArgumentException if <code>bean</code> is null  
1018 \*/  
1019 public PropertyDescriptor[] getPropertyDescriptors(final Object bean) {  
1020  
1021 if (bean == null) {  
1022 throw new IllegalArgumentException("No bean specified");  
1023 }  
1024 return (getPropertyDescriptors(bean.getClass()));  
1025  
1026 }  
1027  
1028  
1029 /\*\*  
1030 \* <p>Return the Java Class repesenting the property editor class that has  
1031 \* been registered for this property (if any). This method follows the  
1032 \* same name resolution rules used by <code>getPropertyDescriptor()</code>,  
1033 \* so if the last element of a name reference is indexed, the property  
1034 \* editor for the underlying property's class is returned.</p>  
1035 \*  
1036 \* <p>Note that <code>null</code> will be returned if there is no property,  
1037 \* or if there is no registered property editor class. Because this  
1038 \* return value is ambiguous, you should determine the existence of the  
1039 \* property itself by other means.</p>  
1040 \*  
1041 \* <p><strong>FIXME</strong> - Does not work with DynaBeans.</p>  
1042 \*  
1043 \* @param bean Bean for which a property descriptor is requested  
1044 \* @param name Possibly indexed and/or nested name of the property for  
1045 \* which a property descriptor is requested  
1046 \* @return the property editor class  
1047 \*  
1048 \* @throws IllegalAccessException if the caller does not have  
1049 \* access to the property accessor method  
1050 \* @throws IllegalArgumentException if <code>bean</code> or  
1051 \* <code>name</code> is null  
1052 \* @throws IllegalArgumentException if a nested reference to a  
1053 \* property returns null  
1054 \* @throws InvocationTargetException if the property accessor method  
1055 \* throws an exception  
1056 \* @throws NoSuchMethodException if an accessor method for this  
1057 \* propety cannot be found  
1058 \*/  
1059 public Class<?> getPropertyEditorClass(final Object bean, final String name)  
1060 throws IllegalAccessException, InvocationTargetException,  
1061 NoSuchMethodException {  
1062  
1063 if (bean == null) {  
1064 throw new IllegalArgumentException("No bean specified");  
1065 }  
1066 if (name == null) {  
1067 throw new IllegalArgumentException("No name specified for bean class '" +  
1068 bean.getClass() + "'");  
1069 }  
1070  
1071 final PropertyDescriptor descriptor =  
1072 getPropertyDescriptor(bean, name);  
1073 if (descriptor != null) {  
1074 return (descriptor.getPropertyEditorClass());  
1075 } else {  
1076 return (null);  
1077 }  
1078  
1079 }  
1080  
1081  
1082 /\*\*  
1083 \* Return the Java Class representing the property type of the specified  
1084 \* property, or <code>null</code> if there is no such property for the  
1085 \* specified bean. This method follows the same name resolution rules  
1086 \* used by <code>getPropertyDescriptor()</code>, so if the last element  
1087 \* of a name reference is indexed, the type of the property itself will  
1088 \* be returned. If the last (or only) element has no property with the  
1089 \* specified name, <code>null</code> is returned.  
1090 \* <p>  
1091 \* If the property is an indexed property (e.g. <code>String[]</code>),  
1092 \* this method will return the type of the items within that array.  
1093 \* Note that from Java 8 and newer, this method do not support  
1094 \* such index types from items within an Collection, and will  
1095 \* instead return the collection type (e.g. java.util.List) from the  
1096 \* getter mtethod.  
1097 \*  
1098 \* @param bean Bean for which a property descriptor is requested  
1099 \* @param name Possibly indexed and/or nested name of the property for  
1100 \* which a property descriptor is requested  
1101 \* @return The property type  
1102 \*  
1103 \* @throws IllegalAccessException if the caller does not have  
1104 \* access to the property accessor method  
1105 \* @throws IllegalArgumentException if <code>bean</code> or  
1106 \* <code>name</code> is null  
1107 \* @throws IllegalArgumentException if a nested reference to a  
1108 \* property returns null  
1109 \* @throws InvocationTargetException if the property accessor method  
1110 \* throws an exception  
1111 \* @throws NoSuchMethodException if an accessor method for this  
1112 \* propety cannot be found  
1113 \*/  
1114 public Class<?> getPropertyType(Object bean, String name)  
1115 throws IllegalAccessException, InvocationTargetException,  
1116 NoSuchMethodException {  
1117  
1118 if (bean == null) {  
1119 throw new IllegalArgumentException("No bean specified");  
1120 }  
1121 if (name == null) {  
1122 throw new IllegalArgumentException("No name specified for bean class '" +  
1123 bean.getClass() + "'");  
1124 }  
1125  
1126 // Resolve nested references  
1127 while (resolver.hasNested(name)) {  
1128 final String next = resolver.next(name);  
1129 final Object nestedBean = getProperty(bean, next);  
1130 if (nestedBean == null) {  
1131 throw new NestedNullException  
1132 ("Null property value for '" + next +  
1133 "' on bean class '" + bean.getClass() + "'");  
1134 }  
1135 bean = nestedBean;  
1136 name = resolver.remove(name);  
1137 }  
1138  
1139 // Remove any subscript from the final name value  
1140 name = resolver.getProperty(name);  
1141  
1142 // Special handling for DynaBeans  
1143 if (bean instanceof DynaBean) {  
1144 final DynaProperty descriptor =  
1145 ((DynaBean) bean).getDynaClass().getDynaProperty(name);  
1146 if (descriptor == null) {  
1147 return (null);  
1148 }  
1149 final Class<?> type = descriptor.getType();  
1150 if (type == null) {  
1151 return (null);  
1152 } else if (type.isArray()) {  
1153 return (type.getComponentType());  
1154 } else {  
1155 return (type);  
1156 }  
1157 }  
1158  
1159 final PropertyDescriptor descriptor =  
1160 getPropertyDescriptor(bean, name);  
1161 if (descriptor == null) {  
1162 return (null);  
1163 } else if (descriptor instanceof IndexedPropertyDescriptor) {  
1164 return (((IndexedPropertyDescriptor) descriptor).  
1165 getIndexedPropertyType());  
1166 } else if (descriptor instanceof MappedPropertyDescriptor) {  
1167 return (((MappedPropertyDescriptor) descriptor).  
1168 getMappedPropertyType());  
1169 } else {  
1170 return (descriptor.getPropertyType());  
1171 }  
1172  
1173 }  
1174  
1175  
1176 /\*\*  
1177 \* <p>Return an accessible property getter method for this property,  
1178 \* if there is one; otherwise return <code>null</code>.</p>  
1179 \*  
1180 \* <p><strong>FIXME</strong> - Does not work with DynaBeans.</p>  
1181 \*  
1182 \* @param descriptor Property descriptor to return a getter for  
1183 \* @return The read method  
1184 \*/  
1185 public Method getReadMethod(final PropertyDescriptor descriptor) {  
1186  
1187 return (MethodUtils.getAccessibleMethod(descriptor.getReadMethod()));  
1188  
1189 }  
1190  
1191  
1192 /\*\*  
1193 \* <p>Return an accessible property getter method for this property,  
1194 \* if there is one; otherwise return <code>null</code>.</p>  
1195 \*  
1196 \* <p><strong>FIXME</strong> - Does not work with DynaBeans.</p>  
1197 \*  
1198 \* @param clazz The class of the read method will be invoked on  
1199 \* @param descriptor Property descriptor to return a getter for  
1200 \* @return The read method  
1201 \*/  
1202 Method getReadMethod(final Class<?> clazz, final PropertyDescriptor descriptor) {  
1203 return (MethodUtils.getAccessibleMethod(clazz, descriptor.getReadMethod()));  
1204 }  
1205  
1206  
1207 /\*\*  
1208 \* Return the value of the specified simple property of the specified  
1209 \* bean, with no type conversions.  
1210 \*  
1211 \* @param bean Bean whose property is to be extracted  
1212 \* @param name Name of the property to be extracted  
1213 \* @return The property value  
1214 \*  
1215 \* @throws IllegalAccessException if the caller does not have  
1216 \* access to the property accessor method  
1217 \* @throws IllegalArgumentException if <code>bean</code> or  
1218 \* <code>name</code> is null  
1219 \* @throws IllegalArgumentException if the property name  
1220 \* is nested or indexed  
1221 \* @throws InvocationTargetException if the property accessor method  
1222 \* throws an exception  
1223 \* @throws NoSuchMethodException if an accessor method for this  
1224 \* propety cannot be found  
1225 \*/  
1226 public Object getSimpleProperty(final Object bean, final String name)  
1227 throws IllegalAccessException, InvocationTargetException,  
1228 NoSuchMethodException {  
1229  
1230 if (bean == null) {  
1231 throw new IllegalArgumentException("No bean specified");  
1232 }  
1233 if (name == null) {  
1234 throw new IllegalArgumentException("No name specified for bean class '" +  
1235 bean.getClass() + "'");  
1236 }  
1237  
1238 // Validate the syntax of the property name  
1239 if (resolver.hasNested(name)) {  
1240 throw new IllegalArgumentException  
1241 ("Nested property names are not allowed: Property '" +  
1242 name + "' on bean class '" + bean.getClass() + "'");  
1243 } else if (resolver.isIndexed(name)) {  
1244 throw new IllegalArgumentException  
1245 ("Indexed property names are not allowed: Property '" +  
1246 name + "' on bean class '" + bean.getClass() + "'");  
1247 } else if (resolver.isMapped(name)) {  
1248 throw new IllegalArgumentException  
1249 ("Mapped property names are not allowed: Property '" +  
1250 name + "' on bean class '" + bean.getClass() + "'");  
1251 }  
1252  
1253 // Handle DynaBean instances specially  
1254 if (bean instanceof DynaBean) {  
1255 final DynaProperty descriptor =  
1256 ((DynaBean) bean).getDynaClass().getDynaProperty(name);  
1257 if (descriptor == null) {  
1258 throw new NoSuchMethodException("Unknown property '" +  
1259 name + "' on dynaclass '" +  
1260 ((DynaBean) bean).getDynaClass() + "'" );  
1261 }  
1262 return (((DynaBean) bean).get(name));  
1263 }  
1264  
1265 // Retrieve the property getter method for the specified property  
1266 final PropertyDescriptor descriptor =  
1267 getPropertyDescriptor(bean, name);  
1268 if (descriptor == null) {  
1269 throw new NoSuchMethodException("Unknown property '" +  
1270 name + "' on class '" + bean.getClass() + "'" );  
1271 }  
1272 final Method readMethod = getReadMethod(bean.getClass(), descriptor);  
1273 if (readMethod == null) {  
1274 throw new NoSuchMethodException("Property '" + name +  
1275 "' has no getter method in class '" + bean.getClass() + "'");  
1276 }  
1277  
1278 // Call the property getter and return the value  
1279 final Object value = invokeMethod(readMethod, bean, EMPTY\_OBJECT\_ARRAY);  
1280 return (value);  
1281  
1282 }  
1283  
1284  
1285 /\*\*  
1286 \* <p>Return an accessible property setter method for this property,  
1287 \* if there is one; otherwise return <code>null</code>.</p>  
1288 \*  
1289 \* <p><em>Note:</em> This method does not work correctly with custom bean  
1290 \* introspection under certain circumstances. It may return {@code null}  
1291 \* even if a write method is defined for the property in question. Use  
1292 \* {@link #getWriteMethod(Class, PropertyDescriptor)} to be sure that the  
1293 \* correct result is returned.</p>  
1294 \* <p><strong>FIXME</strong> - Does not work with DynaBeans.</p>  
1295 \*  
1296 \* @param descriptor Property descriptor to return a setter for  
1297 \* @return The write method  
1298 \*/  
1299 public Method getWriteMethod(final PropertyDescriptor descriptor) {  
1300  
1301 return (MethodUtils.getAccessibleMethod(descriptor.getWriteMethod()));  
1302  
1303 }  
1304  
1305  
1306 /\*\*  
1307 \* <p>Return an accessible property setter method for this property,  
1308 \* if there is one; otherwise return <code>null</code>.</p>  
1309 \*  
1310 \* <p><strong>FIXME</strong> - Does not work with DynaBeans.</p>  
1311 \*  
1312 \* @param clazz The class of the read method will be invoked on  
1313 \* @param descriptor Property descriptor to return a setter for  
1314 \* @return The write method  
1315 \* @since 1.9.1  
1316 \*/  
1317 public Method getWriteMethod(final Class<?> clazz, final PropertyDescriptor descriptor) {  
1318 final BeanIntrospectionData data = getIntrospectionData(clazz);  
1319 return (MethodUtils.getAccessibleMethod(clazz,  
1320 data.getWriteMethod(clazz, descriptor)));  
1321 }  
1322  
1323  
1324 /\*\*  
1325 \* <p>Return <code>true</code> if the specified property name identifies  
1326 \* a readable property on the specified bean; otherwise, return  
1327 \* <code>false</code>.  
1328 \*  
1329 \* @param bean Bean to be examined (may be a {@link DynaBean}  
1330 \* @param name Property name to be evaluated  
1331 \* @return <code>true</code> if the property is readable,  
1332 \* otherwise <code>false</code>  
1333 \*  
1334 \* @throws IllegalArgumentException if <code>bean</code>  
1335 \* or <code>name</code> is <code>null</code>  
1336 \*  
1337 \* @since BeanUtils 1.6  
1338 \*/  
1339 public boolean isReadable(Object bean, String name) {  
1340  
1341 // Validate method parameters  
1342 if (bean == null) {  
1343 throw new IllegalArgumentException("No bean specified");  
1344 }  
1345 if (name == null) {  
1346 throw new IllegalArgumentException("No name specified for bean class '" +  
1347 bean.getClass() + "'");  
1348 }  
1349  
1350 // Resolve nested references  
1351 while (resolver.hasNested(name)) {  
1352 final String next = resolver.next(name);  
1353 Object nestedBean = null;  
1354 try {  
1355 nestedBean = getProperty(bean, next);  
1356 } catch (final IllegalAccessException e) {  
1357 return false;  
1358 } catch (final InvocationTargetException e) {  
1359 return false;  
1360 } catch (final NoSuchMethodException e) {  
1361 return false;  
1362 }  
1363 if (nestedBean == null) {  
1364 throw new NestedNullException  
1365 ("Null property value for '" + next +  
1366 "' on bean class '" + bean.getClass() + "'");  
1367 }  
1368 bean = nestedBean;  
1369 name = resolver.remove(name);  
1370 }  
1371  
1372 // Remove any subscript from the final name value  
1373 name = resolver.getProperty(name);  
1374  
1375 // Treat WrapDynaBean as special case - may be a write-only property  
1376 // (see Jira issue# BEANUTILS-61)  
1377 if (bean instanceof WrapDynaBean) {  
1378 bean = ((WrapDynaBean)bean).getInstance();  
1379 }  
1380  
1381 // Return the requested result  
1382 if (bean instanceof DynaBean) {  
1383 // All DynaBean properties are readable  
1384 return (((DynaBean) bean).getDynaClass().getDynaProperty(name) != null);  
1385 } else {  
1386 try {  
1387 final PropertyDescriptor desc =  
1388 getPropertyDescriptor(bean, name);  
1389 if (desc != null) {  
1390 Method readMethod = getReadMethod(bean.getClass(), desc);  
1391 if (readMethod == null) {  
1392 if (desc instanceof IndexedPropertyDescriptor) {  
1393 readMethod = ((IndexedPropertyDescriptor) desc).getIndexedReadMethod();  
1394 } else if (desc instanceof MappedPropertyDescriptor) {  
1395 readMethod = ((MappedPropertyDescriptor) desc).getMappedReadMethod();  
1396 }  
1397 readMethod = MethodUtils.getAccessibleMethod(bean.getClass(), readMethod);  
1398 }  
1399 return (readMethod != null);  
1400 } else {  
1401 return (false);  
1402 }  
1403 } catch (final IllegalAccessException e) {  
1404 return (false);  
1405 } catch (final InvocationTargetException e) {  
1406 return (false);  
1407 } catch (final NoSuchMethodException e) {  
1408 return (false);  
1409 }  
1410 }  
1411  
1412 }  
1413  
1414  
1415 /\*\*  
1416 \* <p>Return <code>true</code> if the specified property name identifies  
1417 \* a writeable property on the specified bean; otherwise, return  
1418 \* <code>false</code>.  
1419 \*  
1420 \* @param bean Bean to be examined (may be a {@link DynaBean}  
1421 \* @param name Property name to be evaluated  
1422 \* @return <code>true</code> if the property is writeable,  
1423 \* otherwise <code>false</code>  
1424 \*  
1425 \* @throws IllegalArgumentException if <code>bean</code>  
1426 \* or <code>name</code> is <code>null</code>  
1427 \*  
1428 \* @since BeanUtils 1.6  
1429 \*/  
1430 public boolean isWriteable(Object bean, String name) {  
1431  
1432 // Validate method parameters  
1433 if (bean == null) {  
1434 throw new IllegalArgumentException("No bean specified");  
1435 }  
1436 if (name == null) {  
1437 throw new IllegalArgumentException("No name specified for bean class '" +  
1438 bean.getClass() + "'");  
1439 }  
1440  
1441 // Resolve nested references  
1442 while (resolver.hasNested(name)) {  
1443 final String next = resolver.next(name);  
1444 Object nestedBean = null;  
1445 try {  
1446 nestedBean = getProperty(bean, next);  
1447 } catch (final IllegalAccessException e) {  
1448 return false;  
1449 } catch (final InvocationTargetException e) {  
1450 return false;  
1451 } catch (final NoSuchMethodException e) {  
1452 return false;  
1453 }  
1454 if (nestedBean == null) {  
1455 throw new NestedNullException  
1456 ("Null property value for '" + next +  
1457 "' on bean class '" + bean.getClass() + "'");  
1458 }  
1459 bean = nestedBean;  
1460 name = resolver.remove(name);  
1461 }  
1462  
1463 // Remove any subscript from the final name value  
1464 name = resolver.getProperty(name);  
1465  
1466 // Treat WrapDynaBean as special case - may be a read-only property  
1467 // (see Jira issue# BEANUTILS-61)  
1468 if (bean instanceof WrapDynaBean) {  
1469 bean = ((WrapDynaBean)bean).getInstance();  
1470 }  
1471  
1472 // Return the requested result  
1473 if (bean instanceof DynaBean) {  
1474 // All DynaBean properties are writeable  
1475 return (((DynaBean) bean).getDynaClass().getDynaProperty(name) != null);  
1476 } else {  
1477 try {  
1478 final PropertyDescriptor desc =  
1479 getPropertyDescriptor(bean, name);  
1480 if (desc != null) {  
1481 Method writeMethod = getWriteMethod(bean.getClass(), desc);  
1482 if (writeMethod == null) {  
1483 if (desc instanceof IndexedPropertyDescriptor) {  
1484 writeMethod = ((IndexedPropertyDescriptor) desc).getIndexedWriteMethod();  
1485 } else if (desc instanceof MappedPropertyDescriptor) {  
1486 writeMethod = ((MappedPropertyDescriptor) desc).getMappedWriteMethod();  
1487 }  
1488 writeMethod = MethodUtils.getAccessibleMethod(bean.getClass(), writeMethod);  
1489 }  
1490 return (writeMethod != null);  
1491 } else {  
1492 return (false);  
1493 }  
1494 } catch (final IllegalAccessException e) {  
1495 return (false);  
1496 } catch (final InvocationTargetException e) {  
1497 return (false);  
1498 } catch (final NoSuchMethodException e) {  
1499 return (false);  
1500 }  
1501 }  
1502  
1503 }  
1504  
1505  
1506 /\*\*  
1507 \* Set the value of the specified indexed property of the specified  
1508 \* bean, with no type conversions. The zero-relative index of the  
1509 \* required value must be included (in square brackets) as a suffix to  
1510 \* the property name, or <code>IllegalArgumentException</code> will be  
1511 \* thrown. In addition to supporting the JavaBeans specification, this  
1512 \* method has been extended to support <code>List</code> objects as well.  
1513 \*  
1514 \* @param bean Bean whose property is to be modified  
1515 \* @param name <code>propertyname[index]</code> of the property value  
1516 \* to be modified  
1517 \* @param value Value to which the specified property element  
1518 \* should be set  
1519 \*  
1520 \* @throws IndexOutOfBoundsException if the specified index  
1521 \* is outside the valid range for the underlying property  
1522 \* @throws IllegalAccessException if the caller does not have  
1523 \* access to the property accessor method  
1524 \* @throws IllegalArgumentException if <code>bean</code> or  
1525 \* <code>name</code> is null  
1526 \* @throws InvocationTargetException if the property accessor method  
1527 \* throws an exception  
1528 \* @throws NoSuchMethodException if an accessor method for this  
1529 \* propety cannot be found  
1530 \*/  
1531 public void setIndexedProperty(final Object bean, String name,  
1532 final Object value)  
1533 throws IllegalAccessException, InvocationTargetException,  
1534 NoSuchMethodException {  
1535  
1536 if (bean == null) {  
1537 throw new IllegalArgumentException("No bean specified");  
1538 }  
1539 if (name == null) {  
1540 throw new IllegalArgumentException("No name specified for bean class '" +  
1541 bean.getClass() + "'");  
1542 }  
1543  
1544 // Identify the index of the requested individual property  
1545 int index = -1;  
1546 try {  
1547 index = resolver.getIndex(name);  
1548 } catch (final IllegalArgumentException e) {  
1549 throw new IllegalArgumentException("Invalid indexed property '" +  
1550 name + "' on bean class '" + bean.getClass() + "'");  
1551 }  
1552 if (index < 0) {  
1553 throw new IllegalArgumentException("Invalid indexed property '" +  
1554 name + "' on bean class '" + bean.getClass() + "'");  
1555 }  
1556  
1557 // Isolate the name  
1558 name = resolver.getProperty(name);  
1559  
1560 // Set the specified indexed property value  
1561 setIndexedProperty(bean, name, index, value);  
1562  
1563 }  
1564  
1565  
1566 /\*\*  
1567 \* Set the value of the specified indexed property of the specified  
1568 \* bean, with no type conversions. In addition to supporting the JavaBeans  
1569 \* specification, this method has been extended to support  
1570 \* <code>List</code> objects as well.  
1571 \*  
1572 \* @param bean Bean whose property is to be set  
1573 \* @param name Simple property name of the property value to be set  
1574 \* @param index Index of the property value to be set  
1575 \* @param value Value to which the indexed property element is to be set  
1576 \*  
1577 \* @throws IndexOutOfBoundsException if the specified index  
1578 \* is outside the valid range for the underlying property  
1579 \* @throws IllegalAccessException if the caller does not have  
1580 \* access to the property accessor method  
1581 \* @throws IllegalArgumentException if <code>bean</code> or  
1582 \* <code>name</code> is null  
1583 \* @throws InvocationTargetException if the property accessor method  
1584 \* throws an exception  
1585 \* @throws NoSuchMethodException if an accessor method for this  
1586 \* propety cannot be found  
1587 \*/  
1588 public void setIndexedProperty(final Object bean, final String name,  
1589 final int index, final Object value)  
1590 throws IllegalAccessException, InvocationTargetException,  
1591 NoSuchMethodException {  
1592  
1593 if (bean == null) {  
1594 throw new IllegalArgumentException("No bean specified");  
1595 }  
1596 if (name == null || name.length() == 0) {  
1597 if (bean.getClass().isArray()) {  
1598 Array.set(bean, index, value);  
1599 return;  
1600 } else if (bean instanceof List) {  
1601 final List<Object> list = toObjectList(bean);  
1602 list.set(index, value);  
1603 return;  
1604 }  
1605 }  
1606 if (name == null) {  
1607 throw new IllegalArgumentException("No name specified for bean class '" +  
1608 bean.getClass() + "'");  
1609 }  
1610  
1611 // Handle DynaBean instances specially  
1612 if (bean instanceof DynaBean) {  
1613 final DynaProperty descriptor =  
1614 ((DynaBean) bean).getDynaClass().getDynaProperty(name);  
1615 if (descriptor == null) {  
1616 throw new NoSuchMethodException("Unknown property '" +  
1617 name + "' on bean class '" + bean.getClass() + "'");  
1618 }  
1619 ((DynaBean) bean).set(name, index, value);  
1620 return;  
1621 }  
1622  
1623 // Retrieve the property descriptor for the specified property  
1624 final PropertyDescriptor descriptor =  
1625 getPropertyDescriptor(bean, name);  
1626 if (descriptor == null) {  
1627 throw new NoSuchMethodException("Unknown property '" +  
1628 name + "' on bean class '" + bean.getClass() + "'");  
1629 }  
1630  
1631 // Call the indexed setter method if there is one  
1632 if (descriptor instanceof IndexedPropertyDescriptor) {  
1633 Method writeMethod = ((IndexedPropertyDescriptor) descriptor).  
1634 getIndexedWriteMethod();  
1635 writeMethod = MethodUtils.getAccessibleMethod(bean.getClass(), writeMethod);  
1636 if (writeMethod != null) {  
1637 final Object[] subscript = new Object[2];  
1638 subscript[0] = new Integer(index);  
1639 subscript[1] = value;  
1640 try {  
1641 if (log.isTraceEnabled()) {  
1642 final String valueClassName =  
1643 value == null ? "<null>"  
1644 : value.getClass().getName();  
1645 log.trace("setSimpleProperty: Invoking method "  
1646 + writeMethod +" with index=" + index  
1647 + ", value=" + value  
1648 + " (class " + valueClassName+ ")");  
1649 }  
1650 invokeMethod(writeMethod, bean, subscript);  
1651 } catch (final InvocationTargetException e) {  
1652 if (e.getTargetException() instanceof  
1653 IndexOutOfBoundsException) {  
1654 throw (IndexOutOfBoundsException)  
1655 e.getTargetException();  
1656 } else {  
1657 throw e;  
1658 }  
1659 }  
1660 return;  
1661 }  
1662 }  
1663  
1664 // Otherwise, the underlying property must be an array or a list  
1665 final Method readMethod = getReadMethod(bean.getClass(), descriptor);  
1666 if (readMethod == null) {  
1667 throw new NoSuchMethodException("Property '" + name +  
1668 "' has no getter method on bean class '" + bean.getClass() + "'");  
1669 }  
1670  
1671 // Call the property getter to get the array or list  
1672 final Object array = invokeMethod(readMethod, bean, EMPTY\_OBJECT\_ARRAY);  
1673 if (!array.getClass().isArray()) {  
1674 if (array instanceof List) {  
1675 // Modify the specified value in the List  
1676 final List<Object> list = toObjectList(array);  
1677 list.set(index, value);  
1678 } else {  
1679 throw new IllegalArgumentException("Property '" + name +  
1680 "' is not indexed on bean class '" + bean.getClass() + "'");  
1681 }  
1682 } else {  
1683 // Modify the specified value in the array  
1684 Array.set(array, index, value);  
1685 }  
1686  
1687 }  
1688  
1689  
1690 /\*\*  
1691 \* Set the value of the specified mapped property of the  
1692 \* specified bean, with no type conversions. The key of the  
1693 \* value to set must be included (in brackets) as a suffix to  
1694 \* the property name, or <code>IllegalArgumentException</code> will be  
1695 \* thrown.  
1696 \*  
1697 \* @param bean Bean whose property is to be set  
1698 \* @param name <code>propertyname(key)</code> of the property value  
1699 \* to be set  
1700 \* @param value The property value to be set  
1701 \*  
1702 \* @throws IllegalAccessException if the caller does not have  
1703 \* access to the property accessor method  
1704 \* @throws InvocationTargetException if the property accessor method  
1705 \* throws an exception  
1706 \* @throws NoSuchMethodException if an accessor method for this  
1707 \* propety cannot be found  
1708 \*/  
1709 public void setMappedProperty(final Object bean, String name,  
1710 final Object value)  
1711 throws IllegalAccessException, InvocationTargetException,  
1712 NoSuchMethodException {  
1713  
1714 if (bean == null) {  
1715 throw new IllegalArgumentException("No bean specified");  
1716 }  
1717 if (name == null) {  
1718 throw new IllegalArgumentException("No name specified for bean class '" +  
1719 bean.getClass() + "'");  
1720 }  
1721  
1722 // Identify the key of the requested individual property  
1723 String key = null;  
1724 try {  
1725 key = resolver.getKey(name);  
1726 } catch (final IllegalArgumentException e) {  
1727 throw new IllegalArgumentException  
1728 ("Invalid mapped property '" + name +  
1729 "' on bean class '" + bean.getClass() + "'");  
1730 }  
1731 if (key == null) {  
1732 throw new IllegalArgumentException  
1733 ("Invalid mapped property '" + name +  
1734 "' on bean class '" + bean.getClass() + "'");  
1735 }  
1736  
1737 // Isolate the name  
1738 name = resolver.getProperty(name);  
1739  
1740 // Request the specified indexed property value  
1741 setMappedProperty(bean, name, key, value);  
1742  
1743 }  
1744  
1745  
1746 /\*\*  
1747 \* Set the value of the specified mapped property of the specified  
1748 \* bean, with no type conversions.  
1749 \*  
1750 \* @param bean Bean whose property is to be set  
1751 \* @param name Mapped property name of the property value to be set  
1752 \* @param key Key of the property value to be set  
1753 \* @param value The property value to be set  
1754 \*  
1755 \* @throws IllegalAccessException if the caller does not have  
1756 \* access to the property accessor method  
1757 \* @throws InvocationTargetException if the property accessor method  
1758 \* throws an exception  
1759 \* @throws NoSuchMethodException if an accessor method for this  
1760 \* propety cannot be found  
1761 \*/  
1762 public void setMappedProperty(final Object bean, final String name,  
1763 final String key, final Object value)  
1764 throws IllegalAccessException, InvocationTargetException,  
1765 NoSuchMethodException {  
1766  
1767 if (bean == null) {  
1768 throw new IllegalArgumentException("No bean specified");  
1769 }  
1770 if (name == null) {  
1771 throw new IllegalArgumentException("No name specified for bean class '" +  
1772 bean.getClass() + "'");  
1773 }  
1774 if (key == null) {  
1775 throw new IllegalArgumentException("No key specified for property '" +  
1776 name + "' on bean class '" + bean.getClass() + "'");  
1777 }  
1778  
1779 // Handle DynaBean instances specially  
1780 if (bean instanceof DynaBean) {  
1781 final DynaProperty descriptor =  
1782 ((DynaBean) bean).getDynaClass().getDynaProperty(name);  
1783 if (descriptor == null) {  
1784 throw new NoSuchMethodException("Unknown property '" +  
1785 name + "' on bean class '" + bean.getClass() + "'");  
1786 }  
1787 ((DynaBean) bean).set(name, key, value);  
1788 return;  
1789 }  
1790  
1791 // Retrieve the property descriptor for the specified property  
1792 final PropertyDescriptor descriptor =  
1793 getPropertyDescriptor(bean, name);  
1794 if (descriptor == null) {  
1795 throw new NoSuchMethodException("Unknown property '" +  
1796 name + "' on bean class '" + bean.getClass() + "'");  
1797 }  
1798  
1799 if (descriptor instanceof MappedPropertyDescriptor) {  
1800 // Call the keyed setter method if there is one  
1801 Method mappedWriteMethod =  
1802 ((MappedPropertyDescriptor) descriptor).  
1803 getMappedWriteMethod();  
1804 mappedWriteMethod = MethodUtils.getAccessibleMethod(bean.getClass(), mappedWriteMethod);  
1805 if (mappedWriteMethod != null) {  
1806 final Object[] params = new Object[2];  
1807 params[0] = key;  
1808 params[1] = value;  
1809 if (log.isTraceEnabled()) {  
1810 final String valueClassName =  
1811 value == null ? "<null>" : value.getClass().getName();  
1812 log.trace("setSimpleProperty: Invoking method "  
1813 + mappedWriteMethod + " with key=" + key  
1814 + ", value=" + value  
1815 + " (class " + valueClassName +")");  
1816 }  
1817 invokeMethod(mappedWriteMethod, bean, params);  
1818 } else {  
1819 throw new NoSuchMethodException  
1820 ("Property '" + name + "' has no mapped setter method" +  
1821 "on bean class '" + bean.getClass() + "'");  
1822 }  
1823 } else {  
1824 /\* means that the result has to be retrieved from a map \*/  
1825 final Method readMethod = getReadMethod(bean.getClass(), descriptor);  
1826 if (readMethod != null) {  
1827 final Object invokeResult = invokeMethod(readMethod, bean, EMPTY\_OBJECT\_ARRAY);  
1828 /\* test and fetch from the map \*/  
1829 if (invokeResult instanceof java.util.Map) {  
1830 final java.util.Map<String, Object> map = toPropertyMap(invokeResult);  
1831 map.put(key, value);  
1832 }  
1833 } else {  
1834 throw new NoSuchMethodException("Property '" + name +  
1835 "' has no mapped getter method on bean class '" +  
1836 bean.getClass() + "'");  
1837 }  
1838 }  
1839  
1840 }  
1841  
1842  
1843 /\*\*  
1844 \* Set the value of the (possibly nested) property of the specified  
1845 \* name, for the specified bean, with no type conversions.  
1846 \* <p>  
1847 \* Example values for parameter "name" are:  
1848 \* <ul>  
1849 \* <li> "a" -- sets the value of property a of the specified bean </li>  
1850 \* <li> "a.b" -- gets the value of property a of the specified bean,  
1851 \* then on that object sets the value of property b.</li>  
1852 \* <li> "a(key)" -- sets a value of mapped-property a on the specified  
1853 \* bean. This effectively means bean.setA("key").</li>  
1854 \* <li> "a[3]" -- sets a value of indexed-property a on the specified  
1855 \* bean. This effectively means bean.setA(3).</li>  
1856 \* </ul>  
1857 \*  
1858 \* @param bean Bean whose property is to be modified  
1859 \* @param name Possibly nested name of the property to be modified  
1860 \* @param value Value to which the property is to be set  
1861 \*  
1862 \* @throws IllegalAccessException if the caller does not have  
1863 \* access to the property accessor method  
1864 \* @throws IllegalArgumentException if <code>bean</code> or  
1865 \* <code>name</code> is null  
1866 \* @throws IllegalArgumentException if a nested reference to a  
1867 \* property returns null  
1868 \* @throws InvocationTargetException if the property accessor method  
1869 \* throws an exception  
1870 \* @throws NoSuchMethodException if an accessor method for this  
1871 \* propety cannot be found  
1872 \*/  
1873 public void setNestedProperty(Object bean,  
1874 String name, final Object value)  
1875 throws IllegalAccessException, InvocationTargetException,  
1876 NoSuchMethodException {  
1877  
1878 if (bean == null) {  
1879 throw new IllegalArgumentException("No bean specified");  
1880 }  
1881 if (name == null) {  
1882 throw new IllegalArgumentException("No name specified for bean class '" +  
1883 bean.getClass() + "'");  
1884 }  
1885  
1886 // Resolve nested references  
1887 while (resolver.hasNested(name)) {  
1888 final String next = resolver.next(name);  
1889 Object nestedBean = null;  
1890 if (bean instanceof Map) {  
1891 nestedBean = getPropertyOfMapBean((Map<?, ?>)bean, next);  
1892 } else if (resolver.isMapped(next)) {  
1893 nestedBean = getMappedProperty(bean, next);  
1894 } else if (resolver.isIndexed(next)) {  
1895 nestedBean = getIndexedProperty(bean, next);  
1896 } else {  
1897 nestedBean = getSimpleProperty(bean, next);  
1898 }  
1899 if (nestedBean == null) {  
1900 throw new NestedNullException  
1901 ("Null property value for '" + name +  
1902 "' on bean class '" + bean.getClass() + "'");  
1903 }  
1904 bean = nestedBean;  
1905 name = resolver.remove(name);  
1906 }  
1907  
1908 if (bean instanceof Map) {  
1909 setPropertyOfMapBean(toPropertyMap(bean), name, value);  
1910 } else if (resolver.isMapped(name)) {  
1911 setMappedProperty(bean, name, value);  
1912 } else if (resolver.isIndexed(name)) {  
1913 setIndexedProperty(bean, name, value);  
1914 } else {  
1915 setSimpleProperty(bean, name, value);  
1916 }  
1917  
1918 }  
1919  
1920 /\*\*  
1921 \* This method is called by method setNestedProperty when the current bean  
1922 \* is found to be a Map object, and defines how to deal with setting  
1923 \* a property on a Map.  
1924 \* <p>  
1925 \* The standard implementation here is to:  
1926 \* <ul>  
1927 \* <li>call bean.set(propertyName) for all propertyName values.</li>  
1928 \* <li>throw an IllegalArgumentException if the property specifier  
1929 \* contains MAPPED\_DELIM or INDEXED\_DELIM, as Map entries are essentially  
1930 \* simple properties; mapping and indexing operations do not make sense  
1931 \* when accessing a map (even thought the returned object may be a Map  
1932 \* or an Array).</li>  
1933 \* </ul>  
1934 \* <p>  
1935 \* The default behaviour of beanutils 1.7.1 or later is for assigning to  
1936 \* "a.b" to mean a.put(b, obj) always. However the behaviour of beanutils  
1937 \* version 1.6.0, 1.6.1, 1.7.0 was for "a.b" to mean a.setB(obj) if such  
1938 \* a method existed, and a.put(b, obj) otherwise. In version 1.5 it meant  
1939 \* a.put(b, obj) always (ie the same as the behaviour in the current version).  
1940 \* In versions prior to 1.5 it meant a.setB(obj) always. [yes, this is  
1941 \* all <i>very</i> unfortunate]  
1942 \* <p>  
1943 \* Users who would like to customise the meaning of "a.b" in method  
1944 \* setNestedProperty when a is a Map can create a custom subclass of  
1945 \* this class and override this method to implement the behaviour of  
1946 \* their choice, such as restoring the pre-1.4 behaviour of this class  
1947 \* if they wish. When overriding this method, do not forget to deal  
1948 \* with MAPPED\_DELIM and INDEXED\_DELIM characters in the propertyName.  
1949 \* <p>  
1950 \* Note, however, that the recommended solution for objects that  
1951 \* implement Map but want their simple properties to come first is  
1952 \* for <i>those</i> objects to override their get/put methods to implement  
1953 \* that behaviour, and <i>not</i> to solve the problem by modifying the  
1954 \* default behaviour of the PropertyUtilsBean class by overriding this  
1955 \* method.  
1956 \*  
1957 \* @param bean Map bean  
1958 \* @param propertyName The property name  
1959 \* @param value the property value  
1960 \*  
1961 \* @throws IllegalArgumentException when the propertyName is regarded as  
1962 \* being invalid.  
1963 \*  
1964 \* @throws IllegalAccessException just in case subclasses override this  
1965 \* method to try to access real setter methods and find permission is denied.  
1966 \*  
1967 \* @throws InvocationTargetException just in case subclasses override this  
1968 \* method to try to access real setter methods, and find it throws an  
1969 \* exception when invoked.  
1970 \*  
1971 \* @throws NoSuchMethodException just in case subclasses override this  
1972 \* method to try to access real setter methods, and want to fail if  
1973 \* no simple method is available.  
1974 \* @since 1.8.0  
1975 \*/  
1976 protected void setPropertyOfMapBean(final Map<String, Object> bean, String propertyName, final Object value)  
1977 throws IllegalArgumentException, IllegalAccessException,  
1978 InvocationTargetException, NoSuchMethodException {  
1979  
1980 if (resolver.isMapped(propertyName)) {  
1981 final String name = resolver.getProperty(propertyName);  
1982 if (name == null || name.length() == 0) {  
1983 propertyName = resolver.getKey(propertyName);  
1984 }  
1985 }  
1986  
1987 if (resolver.isIndexed(propertyName) ||  
1988 resolver.isMapped(propertyName)) {  
1989 throw new IllegalArgumentException(  
1990 "Indexed or mapped properties are not supported on"  
1991 + " objects of type Map: " + propertyName);  
1992 }  
1993  
1994 bean.put(propertyName, value);  
1995 }  
1996  
1997  
1998  
1999 /\*\*  
2000 \* Set the value of the specified property of the specified bean,  
2001 \* no matter which property reference format is used, with no  
2002 \* type conversions.  
2003 \*  
2004 \* @param bean Bean whose property is to be modified  
2005 \* @param name Possibly indexed and/or nested name of the property  
2006 \* to be modified  
2007 \* @param value Value to which this property is to be set  
2008 \*  
2009 \* @throws IllegalAccessException if the caller does not have  
2010 \* access to the property accessor method  
2011 \* @throws IllegalArgumentException if <code>bean</code> or  
2012 \* <code>name</code> is null  
2013 \* @throws InvocationTargetException if the property accessor method  
2014 \* throws an exception  
2015 \* @throws NoSuchMethodException if an accessor method for this  
2016 \* propety cannot be found  
2017 \*/  
2018 public void setProperty(final Object bean, final String name, final Object value)  
2019 throws IllegalAccessException, InvocationTargetException,  
2020 NoSuchMethodException {  
2021  
2022 setNestedProperty(bean, name, value);  
2023  
2024 }  
2025  
2026  
2027 /\*\*  
2028 \* Set the value of the specified simple property of the specified bean,  
2029 \* with no type conversions.  
2030 \*  
2031 \* @param bean Bean whose property is to be modified  
2032 \* @param name Name of the property to be modified  
2033 \* @param value Value to which the property should be set  
2034 \*  
2035 \* @throws IllegalAccessException if the caller does not have  
2036 \* access to the property accessor method  
2037 \* @throws IllegalArgumentException if <code>bean</code> or  
2038 \* <code>name</code> is null  
2039 \* @throws IllegalArgumentException if the property name is  
2040 \* nested or indexed  
2041 \* @throws InvocationTargetException if the property accessor method  
2042 \* throws an exception  
2043 \* @throws NoSuchMethodException if an accessor method for this  
2044 \* propety cannot be found  
2045 \*/  
2046 public void setSimpleProperty(final Object bean,  
2047 final String name, final Object value)  
2048 throws IllegalAccessException, InvocationTargetException,  
2049 NoSuchMethodException {  
2050  
2051 if (bean == null) {  
2052 throw new IllegalArgumentException("No bean specified");  
2053 }  
2054 if (name == null) {  
2055 throw new IllegalArgumentException("No name specified for bean class '" +  
2056 bean.getClass() + "'");  
2057 }  
2058  
2059 // Validate the syntax of the property name  
2060 if (resolver.hasNested(name)) {  
2061 throw new IllegalArgumentException  
2062 ("Nested property names are not allowed: Property '" +  
2063 name + "' on bean class '" + bean.getClass() + "'");  
2064 } else if (resolver.isIndexed(name)) {  
2065 throw new IllegalArgumentException  
2066 ("Indexed property names are not allowed: Property '" +  
2067 name + "' on bean class '" + bean.getClass() + "'");  
2068 } else if (resolver.isMapped(name)) {  
2069 throw new IllegalArgumentException  
2070 ("Mapped property names are not allowed: Property '" +  
2071 name + "' on bean class '" + bean.getClass() + "'");  
2072 }  
2073  
2074 // Handle DynaBean instances specially  
2075 if (bean instanceof DynaBean) {  
2076 final DynaProperty descriptor =  
2077 ((DynaBean) bean).getDynaClass().getDynaProperty(name);  
2078 if (descriptor == null) {  
2079 throw new NoSuchMethodException("Unknown property '" +  
2080 name + "' on dynaclass '" +  
2081 ((DynaBean) bean).getDynaClass() + "'" );  
2082 }  
2083 ((DynaBean) bean).set(name, value);  
2084 return;  
2085 }  
2086  
2087 // Retrieve the property setter method for the specified property  
2088 final PropertyDescriptor descriptor =  
2089 getPropertyDescriptor(bean, name);  
2090 if (descriptor == null) {  
2091 throw new NoSuchMethodException("Unknown property '" +  
2092 name + "' on class '" + bean.getClass() + "'" );  
2093 }  
2094 final Method writeMethod = getWriteMethod(bean.getClass(), descriptor);  
2095 if (writeMethod == null) {  
2096 throw new NoSuchMethodException("Property '" + name +  
2097 "' has no setter method in class '" + bean.getClass() + "'");  
2098 }  
2099  
2100 // Call the property setter method  
2101 final Object[] values = new Object[1];  
2102 values[0] = value;  
2103 if (log.isTraceEnabled()) {  
2104 final String valueClassName =  
2105 value == null ? "<null>" : value.getClass().getName();  
2106 log.trace("setSimpleProperty: Invoking method " + writeMethod  
2107 + " with value " + value + " (class " + valueClassName + ")");  
2108 }  
2109 invokeMethod(writeMethod, bean, values);  
2110  
2111 }  
2112  
2113 /\*\* This just catches and wraps IllegalArgumentException. \*/  
2114 private Object invokeMethod(  
2115 final Method method,  
2116 final Object bean,  
2117 final Object[] values)  
2118 throws  
2119 IllegalAccessException,  
2120 InvocationTargetException {  
2121 if(bean == null) {  
2122 throw new IllegalArgumentException("No bean specified " +  
2123 "- this should have been checked before reaching this method");  
2124 }  
2125  
2126 try {  
2127  
2128 return method.invoke(bean, values);  
2129  
2130 } catch (final NullPointerException cause) {  
2131 // JDK 1.3 and JDK 1.4 throw NullPointerException if an argument is  
2132 // null for a primitive value (JDK 1.5+ throw IllegalArgumentException)  
2133 String valueString = "";  
2134 if (values != null) {  
2135 for (int i = 0; i < values.length; i++) {  
2136 if (i>0) {  
2137 valueString += ", " ;  
2138 }  
2139 if (values[i] == null) {  
2140 valueString += "<null>";  
2141 } else {  
2142 valueString += (values[i]).getClass().getName();  
2143 }  
2144 }  
2145 }  
2146 String expectedString = "";  
2147 final Class<?>[] parTypes = method.getParameterTypes();  
2148 if (parTypes != null) {  
2149 for (int i = 0; i < parTypes.length; i++) {  
2150 if (i > 0) {  
2151 expectedString += ", ";  
2152 }  
2153 expectedString += parTypes[i].getName();  
2154 }  
2155 }  
2156 final IllegalArgumentException e = new IllegalArgumentException(  
2157 "Cannot invoke " + method.getDeclaringClass().getName() + "."  
2158 + method.getName() + " on bean class '" + bean.getClass() +  
2159 "' - " + cause.getMessage()  
2160 // as per https://issues.apache.org/jira/browse/BEANUTILS-224  
2161 + " - had objects of type \"" + valueString  
2162 + "\" but expected signature \""  
2163 + expectedString + "\""  
2164 );  
2165 if (!BeanUtils.initCause(e, cause)) {  
2166 log.error("Method invocation failed", cause);  
2167 }  
2168 throw e;  
2169 } catch (final IllegalArgumentException cause) {  
2170 String valueString = "";  
2171 if (values != null) {  
2172 for (int i = 0; i < values.length; i++) {  
2173 if (i>0) {  
2174 valueString += ", " ;  
2175 }  
2176 if (values[i] == null) {  
2177 valueString += "<null>";  
2178 } else {  
2179 valueString += (values[i]).getClass().getName();  
2180 }  
2181 }  
2182 }  
2183 String expectedString = "";  
2184 final Class<?>[] parTypes = method.getParameterTypes();  
2185 if (parTypes != null) {  
2186 for (int i = 0; i < parTypes.length; i++) {  
2187 if (i > 0) {  
2188 expectedString += ", ";  
2189 }  
2190 expectedString += parTypes[i].getName();  
2191 }  
2192 }  
2193 final IllegalArgumentException e = new IllegalArgumentException(  
2194 "Cannot invoke " + method.getDeclaringClass().getName() + "."  
2195 + method.getName() + " on bean class '" + bean.getClass() +  
2196 "' - " + cause.getMessage()  
2197 // as per https://issues.apache.org/jira/browse/BEANUTILS-224  
2198 + " - had objects of type \"" + valueString  
2199 + "\" but expected signature \""  
2200 + expectedString + "\""  
2201 );  
2202 if (!BeanUtils.initCause(e, cause)) {  
2203 log.error("Method invocation failed", cause);  
2204 }  
2205 throw e;  
2206  
2207 }  
2208 }  
2209  
2210 /\*\*  
2211 \* Obtains the {@code BeanIntrospectionData} object describing the specified bean  
2212 \* class. This object is looked up in the internal cache. If necessary, introspection  
2213 \* is performed now on the affected bean class, and the results object is created.  
2214 \*  
2215 \* @param beanClass the bean class in question  
2216 \* @return the {@code BeanIntrospectionData} object for this class  
2217 \* @throws IllegalArgumentException if the bean class is <b>null</b>  
2218 \*/  
2219 private BeanIntrospectionData getIntrospectionData(final Class<?> beanClass) {  
2220 if (beanClass == null) {  
2221 throw new IllegalArgumentException("No bean class specified");  
2222 }  
2223  
2224 // Look up any cached information for this bean class  
2225 BeanIntrospectionData data = descriptorsCache.get(beanClass);  
2226 if (data == null) {  
2227 data = fetchIntrospectionData(beanClass);  
2228 descriptorsCache.put(beanClass, data);  
2229 }  
2230  
2231 return data;  
2232 }  
2233  
2234 /\*\*  
2235 \* Performs introspection on the specified class. This method invokes all {@code BeanIntrospector} objects that were  
2236 \* added to this instance.  
2237 \*  
2238 \* @param beanClass the class to be inspected  
2239 \* @return a data object with the results of introspection  
2240 \*/  
2241 private BeanIntrospectionData fetchIntrospectionData(final Class<?> beanClass) {  
2242 final DefaultIntrospectionContext ictx = new DefaultIntrospectionContext(beanClass);  
2243  
2244 for (final BeanIntrospector bi : introspectors) {  
2245 try {  
2246 bi.introspect(ictx);  
2247 } catch (final IntrospectionException iex) {  
2248 log.error("Exception during introspection", iex);  
2249 }  
2250 }  
2251  
2252 return new BeanIntrospectionData(ictx.getPropertyDescriptors());  
2253 }  
2254  
2255 /\*\*  
2256 \* Converts an object to a list of objects. This method is used when dealing  
2257 \* with indexed properties. It assumes that indexed properties are stored as  
2258 \* lists of objects.  
2259 \*  
2260 \* @param obj the object to be converted  
2261 \* @return the resulting list of objects  
2262 \*/  
2263 private static List<Object> toObjectList(final Object obj) {  
2264 @SuppressWarnings("unchecked")  
2265 final  
2266 // indexed properties are stored in lists of objects  
2267 List<Object> list = (List<Object>) obj;  
2268 return list;  
2269 }  
2270  
2271 /\*\*  
2272 \* Converts an object to a map with property values. This method is used  
2273 \* when dealing with mapped properties. It assumes that mapped properties  
2274 \* are stored in a Map<String, Object>.  
2275 \*  
2276 \* @param obj the object to be converted  
2277 \* @return the resulting properties map  
2278 \*/  
2279 private static Map<String, Object> toPropertyMap(final Object obj) {  
2280 @SuppressWarnings("unchecked")  
2281 final  
2282 // mapped properties are stores in maps of type <String, Object>  
2283 Map<String, Object> map = (Map<String, Object>) obj;  
2284 return map;  
2285 }  
2286}