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  
018  
019package org.apache.commons.beanutils;  
020  
021  
022import java.beans.IndexedPropertyDescriptor;  
023import java.beans.PropertyDescriptor;  
024import java.lang.reflect.Array;  
025import java.lang.reflect.InvocationTargetException;  
026import java.lang.reflect.Method;  
027import java.util.ArrayList;  
028import java.util.Collection;  
029import java.util.HashMap;  
030import java.util.List;  
031import java.util.Map;  
032  
033import org.apache.commons.beanutils.expression.Resolver;  
034import org.apache.commons.logging.Log;  
035import org.apache.commons.logging.LogFactory;  
036  
037  
038/\*\*  
039 \* <p>JavaBean property population methods.</p>  
040 \*  
041 \* <p>This class provides implementations for the utility methods in  
042 \* {@link BeanUtils}.  
043 \* Different instances can be used to isolate caches between classloaders  
044 \* and to vary the value converters registered.</p>  
045 \*  
046 \* @version $Id$  
047 \* @see BeanUtils  
048 \* @since 1.7  
049 \*/  
050  
051public class BeanUtilsBean {  
052  
053  
054 // ------------------------------------------------------ Private Class Variables  
055  
056 /\*\*  
057 \* Contains <code>BeanUtilsBean</code> instances indexed by context classloader.  
058 \*/  
059 private static final ContextClassLoaderLocal<BeanUtilsBean>  
060 BEANS\_BY\_CLASSLOADER = new ContextClassLoaderLocal<BeanUtilsBean>() {  
061 // Creates the default instance used when the context classloader is unavailable  
062 @Override  
063 protected BeanUtilsBean initialValue() {  
064 return new BeanUtilsBean();  
065 }  
066 };  
067  
068 /\*\*  
069 \* Gets the instance which provides the functionality for {@link BeanUtils}.  
070 \* This is a pseudo-singleton - an single instance is provided per (thread) context classloader.  
071 \* This mechanism provides isolation for web apps deployed in the same container.  
072 \*  
073 \* @return The (pseudo-singleton) BeanUtils bean instance  
074 \*/  
075 public static BeanUtilsBean getInstance() {  
076 return BEANS\_BY\_CLASSLOADER.get();  
077 }  
078  
079 /\*\*  
080 \* Sets the instance which provides the functionality for {@link BeanUtils}.  
081 \* This is a pseudo-singleton - an single instance is provided per (thread) context classloader.  
082 \* This mechanism provides isolation for web apps deployed in the same container.  
083 \*  
084 \* @param newInstance The (pseudo-singleton) BeanUtils bean instance  
085 \*/  
086 public static void setInstance(final BeanUtilsBean newInstance) {  
087 BEANS\_BY\_CLASSLOADER.set(newInstance);  
088 }  
089  
090 // --------------------------------------------------------- Attributes  
091  
092 /\*\*  
093 \* Logging for this instance  
094 \*/  
095 private final Log log = LogFactory.getLog(BeanUtils.class);  
096  
097 /\*\* Used to perform conversions between object types when setting properties \*/  
098 private final ConvertUtilsBean convertUtilsBean;  
099  
100 /\*\* Used to access properties\*/  
101 private final PropertyUtilsBean propertyUtilsBean;  
102  
103 /\*\* A reference to Throwable's initCause method, or null if it's not there in this JVM \*/  
104 private static final Method INIT\_CAUSE\_METHOD = getInitCauseMethod();  
105  
106 // --------------------------------------------------------- Constuctors  
107  
108 /\*\*  
109 \* <p>Constructs an instance using new property  
110 \* and conversion instances.</p>  
111 \*/  
112 public BeanUtilsBean() {  
113 this(new ConvertUtilsBean(), new PropertyUtilsBean());  
114 }  
115  
116 /\*\*  
117 \* <p>Constructs an instance using given conversion instances  
118 \* and new {@link PropertyUtilsBean} instance.</p>  
119 \*  
120 \* @param convertUtilsBean use this <code>ConvertUtilsBean</code>  
121 \* to perform conversions from one object to another  
122 \*  
123 \* @since 1.8.0  
124 \*/  
125 public BeanUtilsBean(final ConvertUtilsBean convertUtilsBean) {  
126 this(convertUtilsBean, new PropertyUtilsBean());  
127 }  
128  
129 /\*\*  
130 \* <p>Constructs an instance using given property and conversion instances.</p>  
131 \*  
132 \* @param convertUtilsBean use this <code>ConvertUtilsBean</code>  
133 \* to perform conversions from one object to another  
134 \* @param propertyUtilsBean use this <code>PropertyUtilsBean</code>  
135 \* to access properties  
136 \*/  
137 public BeanUtilsBean(  
138 final ConvertUtilsBean convertUtilsBean,  
139 final PropertyUtilsBean propertyUtilsBean) {  
140  
141 this.convertUtilsBean = convertUtilsBean;  
142 this.propertyUtilsBean = propertyUtilsBean;  
143 }  
144  
145 // --------------------------------------------------------- Public Methods  
146  
147 /\*\*  
148 \* <p>Clone a bean based on the available property getters and setters,  
149 \* even if the bean class itself does not implement Cloneable.</p>  
150 \*  
151 \* <p>  
152 \* <strong>Note:</strong> this method creates a <strong>shallow</strong> clone.  
153 \* In other words, any objects referred to by the bean are shared with the clone  
154 \* rather than being cloned in turn.  
155 \* </p>  
156 \*  
157 \* @param bean Bean to be cloned  
158 \* @return the cloned bean  
159 \*  
160 \* @throws IllegalAccessException if the caller does not have  
161 \* access to the property accessor method  
162 \* @throws InstantiationException if a new instance of the bean's  
163 \* class cannot be instantiated  
164 \* @throws InvocationTargetException if the property accessor method  
165 \* throws an exception  
166 \* @throws NoSuchMethodException if an accessor method for this  
167 \* property cannot be found  
168 \*/  
169 public Object cloneBean(final Object bean)  
170 throws IllegalAccessException, InstantiationException,  
171 InvocationTargetException, NoSuchMethodException {  
172  
173 if (log.isDebugEnabled()) {  
174 log.debug("Cloning bean: " + bean.getClass().getName());  
175 }  
176 Object newBean = null;  
177 if (bean instanceof DynaBean) {  
178 newBean = ((DynaBean) bean).getDynaClass().newInstance();  
179 } else {  
180 newBean = bean.getClass().newInstance();  
181 }  
182 getPropertyUtils().copyProperties(newBean, bean);  
183 return (newBean);  
184  
185 }  
186  
187  
188 /\*\*  
189 \* <p>Copy property values from the origin bean to the destination bean  
190 \* for all cases where the property names are the same. For each  
191 \* property, a conversion is attempted as necessary. All combinations of  
192 \* standard JavaBeans and DynaBeans as origin and destination are  
193 \* supported. Properties that exist in the origin bean, but do not exist  
194 \* in the destination bean (or are read-only in the destination bean) are  
195 \* silently ignored.</p>  
196 \*  
197 \* <p>If the origin "bean" is actually a <code>Map</code>, it is assumed  
198 \* to contain String-valued <strong>simple</strong> property names as the keys, pointing at  
199 \* the corresponding property values that will be converted (if necessary)  
200 \* and set in the destination bean. <strong>Note</strong> that this method  
201 \* is intended to perform a "shallow copy" of the properties and so complex  
202 \* properties (for example, nested ones) will not be copied.</p>  
203 \*  
204 \* <p>This method differs from <code>populate()</code>, which  
205 \* was primarily designed for populating JavaBeans from the map of request  
206 \* parameters retrieved on an HTTP request, is that no scalar->indexed  
207 \* or indexed->scalar manipulations are performed. If the origin property  
208 \* is indexed, the destination property must be also.</p>  
209 \*  
210 \* <p>If you know that no type conversions are required, the  
211 \* <code>copyProperties()</code> method in {@link PropertyUtils} will  
212 \* execute faster than this method.</p>  
213 \*  
214 \* <p><strong>FIXME</strong> - Indexed and mapped properties that do not  
215 \* have getter and setter methods for the underlying array or Map are not  
216 \* copied by this method.</p>  
217 \*  
218 \* @param dest Destination bean whose properties are modified  
219 \* @param orig Origin bean whose properties are retrieved  
220 \*  
221 \* @throws IllegalAccessException if the caller does not have  
222 \* access to the property accessor method  
223 \* @throws IllegalArgumentException if the <code>dest</code> or  
224 \* <code>orig</code> argument is null or if the <code>dest</code>  
225 \* property type is different from the source type and the relevant  
226 \* converter has not been registered.  
227 \* @throws InvocationTargetException if the property accessor method  
228 \* throws an exception  
229 \*/  
230 public void copyProperties(final Object dest, final Object orig)  
231 throws IllegalAccessException, InvocationTargetException {  
232  
233 // Validate existence of the specified beans  
234 if (dest == null) {  
235 throw new IllegalArgumentException  
236 ("No destination bean specified");  
237 }  
238 if (orig == null) {  
239 throw new IllegalArgumentException("No origin bean specified");  
240 }  
241 if (log.isDebugEnabled()) {  
242 log.debug("BeanUtils.copyProperties(" + dest + ", " +  
243 orig + ")");  
244 }  
245  
246 // Copy the properties, converting as necessary  
247 if (orig instanceof DynaBean) {  
248 final DynaProperty[] origDescriptors =  
249 ((DynaBean) orig).getDynaClass().getDynaProperties();  
250 for (DynaProperty origDescriptor : origDescriptors) {  
251 final String name = origDescriptor.getName();  
252 // Need to check isReadable() for WrapDynaBean  
253 // (see Jira issue# BEANUTILS-61)  
254 if (getPropertyUtils().isReadable(orig, name) &&  
255 getPropertyUtils().isWriteable(dest, name)) {  
256 final Object value = ((DynaBean) orig).get(name);  
257 copyProperty(dest, name, value);  
258 }  
259 }  
260 } else if (orig instanceof Map) {  
261 @SuppressWarnings("unchecked")  
262 final  
263 // Map properties are always of type <String, Object>  
264 Map<String, Object> propMap = (Map<String, Object>) orig;  
265 for (final Map.Entry<String, Object> entry : propMap.entrySet()) {  
266 final String name = entry.getKey();  
267 if (getPropertyUtils().isWriteable(dest, name)) {  
268 copyProperty(dest, name, entry.getValue());  
269 }  
270 }  
271 } else /\* if (orig is a standard JavaBean) \*/ {  
272 final PropertyDescriptor[] origDescriptors =  
273 getPropertyUtils().getPropertyDescriptors(orig);  
274 for (PropertyDescriptor origDescriptor : origDescriptors) {  
275 final String name = origDescriptor.getName();  
276 if ("class".equals(name)) {  
277 continue; // No point in trying to set an object's class  
278 }  
279 if (getPropertyUtils().isReadable(orig, name) &&  
280 getPropertyUtils().isWriteable(dest, name)) {  
281 try {  
282 final Object value =  
283 getPropertyUtils().getSimpleProperty(orig, name);  
284 copyProperty(dest, name, value);  
285 } catch (final NoSuchMethodException e) {  
286 // Should not happen  
287 }  
288 }  
289 }  
290 }  
291  
292 }  
293  
294  
295 /\*\*  
296 \* <p>Copy the specified property value to the specified destination bean,  
297 \* performing any type conversion that is required. If the specified  
298 \* bean does not have a property of the specified name, or the property  
299 \* is read only on the destination bean, return without  
300 \* doing anything. If you have custom destination property types, register  
301 \* {@link Converter}s for them by calling the <code>register()</code>  
302 \* method of {@link ConvertUtils}.</p>  
303 \*  
304 \* <p><strong>IMPLEMENTATION RESTRICTIONS</strong>:</p>  
305 \* <ul>  
306 \* <li>Does not support destination properties that are indexed,  
307 \* but only an indexed setter (as opposed to an array setter)  
308 \* is available.</li>  
309 \* <li>Does not support destination properties that are mapped,  
310 \* but only a keyed setter (as opposed to a Map setter)  
311 \* is available.</li>  
312 \* <li>The desired property type of a mapped setter cannot be  
313 \* determined (since Maps support any data type), so no conversion  
314 \* will be performed.</li>  
315 \* </ul>  
316 \*  
317 \* @param bean Bean on which setting is to be performed  
318 \* @param name Property name (can be nested/indexed/mapped/combo)  
319 \* @param value Value to be set  
320 \*  
321 \* @throws IllegalAccessException if the caller does not have  
322 \* access to the property accessor method  
323 \* @throws InvocationTargetException if the property accessor method  
324 \* throws an exception  
325 \*/  
326 public void copyProperty(final Object bean, String name, Object value)  
327 throws IllegalAccessException, InvocationTargetException {  
328  
329 // Trace logging (if enabled)  
330 if (log.isTraceEnabled()) {  
331 final StringBuilder sb = new StringBuilder(" copyProperty(");  
332 sb.append(bean);  
333 sb.append(", ");  
334 sb.append(name);  
335 sb.append(", ");  
336 if (value == null) {  
337 sb.append("<NULL>");  
338 } else if (value instanceof String) {  
339 sb.append((String) value);  
340 } else if (value instanceof String[]) {  
341 final String[] values = (String[]) value;  
342 sb.append('[');  
343 for (int i = 0; i < values.length; i++) {  
344 if (i > 0) {  
345 sb.append(',');  
346 }  
347 sb.append(values[i]);  
348 }  
349 sb.append(']');  
350 } else {  
351 sb.append(value.toString());  
352 }  
353 sb.append(')');  
354 log.trace(sb.toString());  
355 }  
356  
357 // Resolve any nested expression to get the actual target bean  
358 Object target = bean;  
359 final Resolver resolver = getPropertyUtils().getResolver();  
360 while (resolver.hasNested(name)) {  
361 try {  
362 target = getPropertyUtils().getProperty(target, resolver.next(name));  
363 name = resolver.remove(name);  
364 } catch (final NoSuchMethodException e) {  
365 return; // Skip this property setter  
366 }  
367 }  
368 if (log.isTraceEnabled()) {  
369 log.trace(" Target bean = " + target);  
370 log.trace(" Target name = " + name);  
371 }  
372  
373 // Declare local variables we will require  
374 final String propName = resolver.getProperty(name); // Simple name of target property  
375 Class<?> type = null; // Java type of target property  
376 final int index = resolver.getIndex(name); // Indexed subscript value (if any)  
377 final String key = resolver.getKey(name); // Mapped key value (if any)  
378  
379 // Calculate the target property type  
380 if (target instanceof DynaBean) {  
381 final DynaClass dynaClass = ((DynaBean) target).getDynaClass();  
382 final DynaProperty dynaProperty = dynaClass.getDynaProperty(propName);  
383 if (dynaProperty == null) {  
384 return; // Skip this property setter  
385 }  
386 type = dynaPropertyType(dynaProperty, value);  
387 } else {  
388 PropertyDescriptor descriptor = null;  
389 try {  
390 descriptor =  
391 getPropertyUtils().getPropertyDescriptor(target, name);  
392 if (descriptor == null) {  
393 return; // Skip this property setter  
394 }  
395 } catch (final NoSuchMethodException e) {  
396 return; // Skip this property setter  
397 }  
398 type = descriptor.getPropertyType();  
399 if (type == null) {  
400 // Most likely an indexed setter on a POJB only  
401 if (log.isTraceEnabled()) {  
402 log.trace(" target type for property '" +  
403 propName + "' is null, so skipping ths setter");  
404 }  
405 return;  
406 }  
407 }  
408 if (log.isTraceEnabled()) {  
409 log.trace(" target propName=" + propName + ", type=" +  
410 type + ", index=" + index + ", key=" + key);  
411 }  
412  
413 // Convert the specified value to the required type and store it  
414 if (index >= 0) { // Destination must be indexed  
415 value = convertForCopy(value, type.getComponentType());  
416 try {  
417 getPropertyUtils().setIndexedProperty(target, propName,  
418 index, value);  
419 } catch (final NoSuchMethodException e) {  
420 throw new InvocationTargetException  
421 (e, "Cannot set " + propName);  
422 }  
423 } else if (key != null) { // Destination must be mapped  
424 // Maps do not know what the preferred data type is,  
425 // so perform no conversions at all  
426 // FIXME - should we create or support a TypedMap?  
427 try {  
428 getPropertyUtils().setMappedProperty(target, propName,  
429 key, value);  
430 } catch (final NoSuchMethodException e) {  
431 throw new InvocationTargetException  
432 (e, "Cannot set " + propName);  
433 }  
434 } else { // Destination must be simple  
435 value = convertForCopy(value, type);  
436 try {  
437 getPropertyUtils().setSimpleProperty(target, propName, value);  
438 } catch (final NoSuchMethodException e) {  
439 throw new InvocationTargetException  
440 (e, "Cannot set " + propName);  
441 }  
442 }  
443  
444 }  
445  
446  
447 /\*\*  
448 \* <p>Return the entire set of properties for which the specified bean  
449 \* provides a read method. This map contains the to <code>String</code>  
450 \* converted property values for all properties for which a read method  
451 \* is provided (i.e. where the getReadMethod() returns non-null).</p>  
452 \*  
453 \* <p>This map can be fed back to a call to  
454 \* <code>BeanUtils.populate()</code> to reconsitute the same set of  
455 \* properties, modulo differences for read-only and write-only  
456 \* properties, but only if there are no indexed properties.</p>  
457 \*  
458 \* <p><strong>Warning:</strong> if any of the bean property implementations  
459 \* contain (directly or indirectly) a call to this method then  
460 \* a stack overflow may result. For example:  
461 \* <code><pre>  
462 \* class MyBean  
463 \* {  
464 \* public Map getParameterMap()  
465 \* {  
466 \* BeanUtils.describe(this);  
467 \* }  
468 \* }  
469 \* </pre></code>  
470 \* will result in an infinite regression when <code>getParametersMap</code>  
471 \* is called. It is recommended that such methods are given alternative  
472 \* names (for example, <code>parametersMap</code>).  
473 \* </p>  
474 \* @param bean Bean whose properties are to be extracted  
475 \* @return Map of property descriptors  
476 \*  
477 \* @throws IllegalAccessException if the caller does not have  
478 \* access to the property accessor method  
479 \* @throws InvocationTargetException if the property accessor method  
480 \* throws an exception  
481 \* @throws NoSuchMethodException if an accessor method for this  
482 \* property cannot be found  
483 \*/  
484 public Map<String, String> describe(final Object bean)  
485 throws IllegalAccessException, InvocationTargetException,  
486 NoSuchMethodException {  
487  
488 if (bean == null) {  
489 // return (Collections.EMPTY\_MAP);  
490 return (new java.util.HashMap<String, String>());  
491 }  
492  
493 if (log.isDebugEnabled()) {  
494 log.debug("Describing bean: " + bean.getClass().getName());  
495 }  
496  
497 final Map<String, String> description = new HashMap<String, String>();  
498 if (bean instanceof DynaBean) {  
499 final DynaProperty[] descriptors =  
500 ((DynaBean) bean).getDynaClass().getDynaProperties();  
501 for (DynaProperty descriptor : descriptors) {  
502 final String name = descriptor.getName();  
503 description.put(name, getProperty(bean, name));  
504 }  
505 } else {  
506 final PropertyDescriptor[] descriptors =  
507 getPropertyUtils().getPropertyDescriptors(bean);  
508 final Class<?> clazz = bean.getClass();  
509 for (PropertyDescriptor descriptor : descriptors) {  
510 final String name = descriptor.getName();  
511 if (getPropertyUtils().getReadMethod(clazz, descriptor) != null) {  
512 description.put(name, getProperty(bean, name));  
513 }  
514 }  
515 }  
516 return (description);  
517  
518 }  
519  
520  
521 /\*\*  
522 \* Return the value of the specified array property of the specified  
523 \* bean, as a String array.  
524 \*  
525 \* @param bean Bean whose property is to be extracted  
526 \* @param name Name of the property to be extracted  
527 \* @return The array property value  
528 \*  
529 \* @throws IllegalAccessException if the caller does not have  
530 \* access to the property accessor method  
531 \* @throws InvocationTargetException if the property accessor method  
532 \* throws an exception  
533 \* @throws NoSuchMethodException if an accessor method for this  
534 \* property cannot be found  
535 \*/  
536 public String[] getArrayProperty(final Object bean, final String name)  
537 throws IllegalAccessException, InvocationTargetException,  
538 NoSuchMethodException {  
539  
540 final Object value = getPropertyUtils().getProperty(bean, name);  
541 if (value == null) {  
542 return (null);  
543 } else if (value instanceof Collection) {  
544 final ArrayList<String> values = new ArrayList<String>();  
545 for (final Object item : (Collection<?>) value) {  
546 if (item == null) {  
547 values.add(null);  
548 } else {  
549 // convert to string using convert utils  
550 values.add(getConvertUtils().convert(item));  
551 }  
552 }  
553 return (values.toArray(new String[values.size()]));  
554 } else if (value.getClass().isArray()) {  
555 final int n = Array.getLength(value);  
556 final String[] results = new String[n];  
557 for (int i = 0; i < n; i++) {  
558 final Object item = Array.get(value, i);  
559 if (item == null) {  
560 results[i] = null;  
561 } else {  
562 // convert to string using convert utils  
563 results[i] = getConvertUtils().convert(item);  
564 }  
565 }  
566 return (results);  
567 } else {  
568 final String[] results = new String[1];  
569 results[0] = getConvertUtils().convert(value);  
570 return (results);  
571 }  
572  
573 }  
574  
575  
576 /\*\*  
577 \* Return the value of the specified indexed property of the specified  
578 \* bean, as a String. The zero-relative index of the  
579 \* required value must be included (in square brackets) as a suffix to  
580 \* the property name, or <code>IllegalArgumentException</code> will be  
581 \* thrown.  
582 \*  
583 \* @param bean Bean whose property is to be extracted  
584 \* @param name <code>propertyname[index]</code> of the property value  
585 \* to be extracted  
586 \* @return The indexed property's value, converted to a String  
587 \*  
588 \* @throws IllegalAccessException if the caller does not have  
589 \* access to the property accessor method  
590 \* @throws InvocationTargetException if the property accessor method  
591 \* throws an exception  
592 \* @throws NoSuchMethodException if an accessor method for this  
593 \* property cannot be found  
594 \*/  
595 public String getIndexedProperty(final Object bean, final String name)  
596 throws IllegalAccessException, InvocationTargetException,  
597 NoSuchMethodException {  
598  
599 final Object value = getPropertyUtils().getIndexedProperty(bean, name);  
600 return (getConvertUtils().convert(value));  
601  
602 }  
603  
604  
605 /\*\*  
606 \* Return the value of the specified indexed property of the specified  
607 \* bean, as a String. The index is specified as a method parameter and  
608 \* must \*not\* be included in the property name expression  
609 \*  
610 \* @param bean Bean whose property is to be extracted  
611 \* @param name Simple property name of the property value to be extracted  
612 \* @param index Index of the property value to be extracted  
613 \* @return The indexed property's value, converted to a String  
614 \*  
615 \* @throws IllegalAccessException if the caller does not have  
616 \* access to the property accessor method  
617 \* @throws InvocationTargetException if the property accessor method  
618 \* throws an exception  
619 \* @throws NoSuchMethodException if an accessor method for this  
620 \* property cannot be found  
621 \*/  
622 public String getIndexedProperty(final Object bean,  
623 final String name, final int index)  
624 throws IllegalAccessException, InvocationTargetException,  
625 NoSuchMethodException {  
626  
627 final Object value = getPropertyUtils().getIndexedProperty(bean, name, index);  
628 return (getConvertUtils().convert(value));  
629  
630 }  
631  
632  
633 /\*\*  
634 \* Return the value of the specified indexed property of the specified  
635 \* bean, as a String. The String-valued key of the required value  
636 \* must be included (in parentheses) as a suffix to  
637 \* the property name, or <code>IllegalArgumentException</code> will be  
638 \* thrown.  
639 \*  
640 \* @param bean Bean whose property is to be extracted  
641 \* @param name <code>propertyname(index)</code> of the property value  
642 \* to be extracted  
643 \* @return The mapped property's value, converted to a String  
644 \*  
645 \* @throws IllegalAccessException if the caller does not have  
646 \* access to the property accessor method  
647 \* @throws InvocationTargetException if the property accessor method  
648 \* throws an exception  
649 \* @throws NoSuchMethodException if an accessor method for this  
650 \* property cannot be found  
651 \*/  
652 public String getMappedProperty(final Object bean, final String name)  
653 throws IllegalAccessException, InvocationTargetException,  
654 NoSuchMethodException {  
655  
656 final Object value = getPropertyUtils().getMappedProperty(bean, name);  
657 return (getConvertUtils().convert(value));  
658  
659 }  
660  
661  
662 /\*\*  
663 \* Return the value of the specified mapped property of the specified  
664 \* bean, as a String. The key is specified as a method parameter and  
665 \* must \*not\* be included in the property name expression  
666 \*  
667 \* @param bean Bean whose property is to be extracted  
668 \* @param name Simple property name of the property value to be extracted  
669 \* @param key Lookup key of the property value to be extracted  
670 \* @return The mapped property's value, converted to a String  
671 \*  
672 \* @throws IllegalAccessException if the caller does not have  
673 \* access to the property accessor method  
674 \* @throws InvocationTargetException if the property accessor method  
675 \* throws an exception  
676 \* @throws NoSuchMethodException if an accessor method for this  
677 \* property cannot be found  
678 \*/  
679 public String getMappedProperty(final Object bean,  
680 final String name, final String key)  
681 throws IllegalAccessException, InvocationTargetException,  
682 NoSuchMethodException {  
683  
684 final Object value = getPropertyUtils().getMappedProperty(bean, name, key);  
685 return (getConvertUtils().convert(value));  
686  
687 }  
688  
689  
690 /\*\*  
691 \* Return the value of the (possibly nested) property of the specified  
692 \* name, for the specified bean, as a String.  
693 \*  
694 \* @param bean Bean whose property is to be extracted  
695 \* @param name Possibly nested name of the property to be extracted  
696 \* @return The nested property's value, converted to a String  
697 \*  
698 \* @throws IllegalAccessException if the caller does not have  
699 \* access to the property accessor method  
700 \* @throws IllegalArgumentException if a nested reference to a  
701 \* property returns null  
702 \* @throws InvocationTargetException if the property accessor method  
703 \* throws an exception  
704 \* @throws NoSuchMethodException if an accessor method for this  
705 \* property cannot be found  
706 \*/  
707 public String getNestedProperty(final Object bean, final String name)  
708 throws IllegalAccessException, InvocationTargetException,  
709 NoSuchMethodException {  
710  
711 final Object value = getPropertyUtils().getNestedProperty(bean, name);  
712 return (getConvertUtils().convert(value));  
713  
714 }  
715  
716  
717 /\*\*  
718 \* Return the value of the specified property of the specified bean,  
719 \* no matter which property reference format is used, as a String.  
720 \*  
721 \* @param bean Bean whose property is to be extracted  
722 \* @param name Possibly indexed and/or nested name of the property  
723 \* to be extracted  
724 \* @return The property's value, converted to a String  
725 \*  
726 \* @throws IllegalAccessException if the caller does not have  
727 \* access to the property accessor method  
728 \* @throws InvocationTargetException if the property accessor method  
729 \* throws an exception  
730 \* @throws NoSuchMethodException if an accessor method for this  
731 \* property cannot be found  
732 \*/  
733 public String getProperty(final Object bean, final String name)  
734 throws IllegalAccessException, InvocationTargetException,  
735 NoSuchMethodException {  
736  
737 return (getNestedProperty(bean, name));  
738  
739 }  
740  
741  
742 /\*\*  
743 \* Return the value of the specified simple property of the specified  
744 \* bean, converted to a String.  
745 \*  
746 \* @param bean Bean whose property is to be extracted  
747 \* @param name Name of the property to be extracted  
748 \* @return The property's value, converted to a String  
749 \*  
750 \* @throws IllegalAccessException if the caller does not have  
751 \* access to the property accessor method  
752 \* @throws InvocationTargetException if the property accessor method  
753 \* throws an exception  
754 \* @throws NoSuchMethodException if an accessor method for this  
755 \* property cannot be found  
756 \*/  
757 public String getSimpleProperty(final Object bean, final String name)  
758 throws IllegalAccessException, InvocationTargetException,  
759 NoSuchMethodException {  
760  
761 final Object value = getPropertyUtils().getSimpleProperty(bean, name);  
762 return (getConvertUtils().convert(value));  
763  
764 }  
765  
766  
767 /\*\*  
768 \* <p>Populate the JavaBeans properties of the specified bean, based on  
769 \* the specified name/value pairs. This method uses Java reflection APIs  
770 \* to identify corresponding "property setter" method names, and deals  
771 \* with setter arguments of type <code>String</code>, <code>boolean</code>,  
772 \* <code>int</code>, <code>long</code>, <code>float</code>, and  
773 \* <code>double</code>. In addition, array setters for these types (or the  
774 \* corresponding primitive types) can also be identified.</p>  
775 \*  
776 \* <p>The particular setter method to be called for each property is  
777 \* determined using the usual JavaBeans introspection mechanisms. Thus,  
778 \* you may identify custom setter methods using a BeanInfo class that is  
779 \* associated with the class of the bean itself. If no such BeanInfo  
780 \* class is available, the standard method name conversion ("set" plus  
781 \* the capitalized name of the property in question) is used.</p>  
782 \*  
783 \* <p><strong>NOTE</strong>: It is contrary to the JavaBeans Specification  
784 \* to have more than one setter method (with different argument  
785 \* signatures) for the same property.</p>  
786 \*  
787 \* <p><strong>WARNING</strong> - The logic of this method is customized  
788 \* for extracting String-based request parameters from an HTTP request.  
789 \* It is probably not what you want for general property copying with  
790 \* type conversion. For that purpose, check out the  
791 \* <code>copyProperties()</code> method instead.</p>  
792 \*  
793 \* @param bean JavaBean whose properties are being populated  
794 \* @param properties Map keyed by property name, with the  
795 \* corresponding (String or String[]) value(s) to be set  
796 \*  
797 \* @throws IllegalAccessException if the caller does not have  
798 \* access to the property accessor method  
799 \* @throws InvocationTargetException if the property accessor method  
800 \* throws an exception  
801 \*/  
802 public void populate(final Object bean, final Map<String, ? extends Object> properties)  
803 throws IllegalAccessException, InvocationTargetException {  
804  
805 // Do nothing unless both arguments have been specified  
806 if ((bean == null) || (properties == null)) {  
807 return;  
808 }  
809 if (log.isDebugEnabled()) {  
810 log.debug("BeanUtils.populate(" + bean + ", " +  
811 properties + ")");  
812 }  
813  
814 // Loop through the property name/value pairs to be set  
815 for(final Map.Entry<String, ? extends Object> entry : properties.entrySet()) {  
816 // Identify the property name and value(s) to be assigned  
817 final String name = entry.getKey();  
818 if (name == null) {  
819 continue;  
820 }  
821  
822 // Perform the assignment for this property  
823 setProperty(bean, name, entry.getValue());  
824  
825 }  
826  
827 }  
828  
829  
830 /\*\*  
831 \* <p>Set the specified property value, performing type conversions as  
832 \* required to conform to the type of the destination property.</p>  
833 \*  
834 \* <p>If the property is read only then the method returns  
835 \* without throwing an exception.</p>  
836 \*  
837 \* <p>If <code>null</code> is passed into a property expecting a primitive value,  
838 \* then this will be converted as if it were a <code>null</code> string.</p>  
839 \*  
840 \* <p><strong>WARNING</strong> - The logic of this method is customized  
841 \* to meet the needs of <code>populate()</code>, and is probably not what  
842 \* you want for general property copying with type conversion. For that  
843 \* purpose, check out the <code>copyProperty()</code> method instead.</p>  
844 \*  
845 \* <p><strong>WARNING</strong> - PLEASE do not modify the behavior of this  
846 \* method without consulting with the Struts developer community. There  
847 \* are some subtleties to its functionality that are not documented in the  
848 \* Javadoc description above, yet are vital to the way that Struts utilizes  
849 \* this method.</p>  
850 \*  
851 \* @param bean Bean on which setting is to be performed  
852 \* @param name Property name (can be nested/indexed/mapped/combo)  
853 \* @param value Value to be set  
854 \*  
855 \* @throws IllegalAccessException if the caller does not have  
856 \* access to the property accessor method  
857 \* @throws InvocationTargetException if the property accessor method  
858 \* throws an exception  
859 \*/  
860 public void setProperty(final Object bean, String name, final Object value)  
861 throws IllegalAccessException, InvocationTargetException {  
862  
863 // Trace logging (if enabled)  
864 if (log.isTraceEnabled()) {  
865 final StringBuilder sb = new StringBuilder(" setProperty(");  
866 sb.append(bean);  
867 sb.append(", ");  
868 sb.append(name);  
869 sb.append(", ");  
870 if (value == null) {  
871 sb.append("<NULL>");  
872 } else if (value instanceof String) {  
873 sb.append((String) value);  
874 } else if (value instanceof String[]) {  
875 final String[] values = (String[]) value;  
876 sb.append('[');  
877 for (int i = 0; i < values.length; i++) {  
878 if (i > 0) {  
879 sb.append(',');  
880 }  
881 sb.append(values[i]);  
882 }  
883 sb.append(']');  
884 } else {  
885 sb.append(value.toString());  
886 }  
887 sb.append(')');  
888 log.trace(sb.toString());  
889 }  
890  
891 // Resolve any nested expression to get the actual target bean  
892 Object target = bean;  
893 final Resolver resolver = getPropertyUtils().getResolver();  
894 while (resolver.hasNested(name)) {  
895 try {  
896 target = getPropertyUtils().getProperty(target, resolver.next(name));  
897 if (target == null) { // the value of a nested property is null  
898 return;  
899 }  
900 name = resolver.remove(name);  
901 } catch (final NoSuchMethodException e) {  
902 return; // Skip this property setter  
903 }  
904 }  
905 if (log.isTraceEnabled()) {  
906 log.trace(" Target bean = " + target);  
907 log.trace(" Target name = " + name);  
908 }  
909  
910 // Declare local variables we will require  
911 final String propName = resolver.getProperty(name); // Simple name of target property  
912 Class<?> type = null; // Java type of target property  
913 final int index = resolver.getIndex(name); // Indexed subscript value (if any)  
914 final String key = resolver.getKey(name); // Mapped key value (if any)  
915  
916 // Calculate the property type  
917 if (target instanceof DynaBean) {  
918 final DynaClass dynaClass = ((DynaBean) target).getDynaClass();  
919 final DynaProperty dynaProperty = dynaClass.getDynaProperty(propName);  
920 if (dynaProperty == null) {  
921 return; // Skip this property setter  
922 }  
923 type = dynaPropertyType(dynaProperty, value);  
924 if (index >= 0 && List.class.isAssignableFrom(type)) {  
925 type = Object.class;  
926 }  
927 } else if (target instanceof Map) {  
928 type = Object.class;  
929 } else if (target != null && target.getClass().isArray() && index >= 0) {  
930 type = Array.get(target, index).getClass();  
931 } else {  
932 PropertyDescriptor descriptor = null;  
933 try {  
934 descriptor =  
935 getPropertyUtils().getPropertyDescriptor(target, name);  
936 if (descriptor == null) {  
937 return; // Skip this property setter  
938 }  
939 } catch (final NoSuchMethodException e) {  
940 return; // Skip this property setter  
941 }  
942 if (descriptor instanceof MappedPropertyDescriptor) {  
943 if (((MappedPropertyDescriptor) descriptor).getMappedWriteMethod() == null) {  
944 if (log.isDebugEnabled()) {  
945 log.debug("Skipping read-only property");  
946 }  
947 return; // Read-only, skip this property setter  
948 }  
949 type = ((MappedPropertyDescriptor) descriptor).  
950 getMappedPropertyType();  
951 } else if (index >= 0 && descriptor instanceof IndexedPropertyDescriptor) {  
952 if (((IndexedPropertyDescriptor) descriptor).getIndexedWriteMethod() == null) {  
953 if (log.isDebugEnabled()) {  
954 log.debug("Skipping read-only property");  
955 }  
956 return; // Read-only, skip this property setter  
957 }  
958 type = ((IndexedPropertyDescriptor) descriptor).  
959 getIndexedPropertyType();  
960 } else if (index >= 0 && List.class.isAssignableFrom(descriptor.getPropertyType())) {  
961 type = Object.class;  
962 } else if (key != null) {  
963 if (descriptor.getReadMethod() == null) {  
964 if (log.isDebugEnabled()) {  
965 log.debug("Skipping read-only property");  
966 }  
967 return; // Read-only, skip this property setter  
968 }  
969 type = (value == null) ? Object.class : value.getClass();  
970 } else {  
971 if (descriptor.getWriteMethod() == null) {  
972 if (log.isDebugEnabled()) {  
973 log.debug("Skipping read-only property");  
974 }  
975 return; // Read-only, skip this property setter  
976 }  
977 type = descriptor.getPropertyType();  
978 }  
979 }  
980  
981 // Convert the specified value to the required type  
982 Object newValue = null;  
983 if (type.isArray() && (index < 0)) { // Scalar value into array  
984 if (value == null) {  
985 final String[] values = new String[1];  
986 values[0] = null;  
987 newValue = getConvertUtils().convert(values, type);  
988 } else if (value instanceof String) {  
989 newValue = getConvertUtils().convert(value, type);  
990 } else if (value instanceof String[]) {  
991 newValue = getConvertUtils().convert((String[]) value, type);  
992 } else {  
993 newValue = convert(value, type);  
994 }  
995 } else if (type.isArray()) { // Indexed value into array  
996 if (value instanceof String || value == null) {  
997 newValue = getConvertUtils().convert((String) value,  
998 type.getComponentType());  
999 } else if (value instanceof String[]) {  
1000 newValue = getConvertUtils().convert(((String[]) value)[0],  
1001 type.getComponentType());  
1002 } else {  
1003 newValue = convert(value, type.getComponentType());  
1004 }  
1005 } else { // Value into scalar  
1006 if (value instanceof String) {  
1007 newValue = getConvertUtils().convert((String) value, type);  
1008 } else if (value instanceof String[]) {  
1009 newValue = getConvertUtils().convert(((String[]) value)[0],  
1010 type);  
1011 } else {  
1012 newValue = convert(value, type);  
1013 }  
1014 }  
1015  
1016 // Invoke the setter method  
1017 try {  
1018 getPropertyUtils().setProperty(target, name, newValue);  
1019 } catch (final NoSuchMethodException e) {  
1020 throw new InvocationTargetException  
1021 (e, "Cannot set " + propName);  
1022 }  
1023  
1024 }  
1025  
1026 /\*\*  
1027 \* Gets the <code>ConvertUtilsBean</code> instance used to perform the conversions.  
1028 \*  
1029 \* @return The ConvertUtils bean instance  
1030 \*/  
1031 public ConvertUtilsBean getConvertUtils() {  
1032 return convertUtilsBean;  
1033 }  
1034  
1035 /\*\*  
1036 \* Gets the <code>PropertyUtilsBean</code> instance used to access properties.  
1037 \*  
1038 \* @return The ConvertUtils bean instance  
1039 \*/  
1040 public PropertyUtilsBean getPropertyUtils() {  
1041 return propertyUtilsBean;  
1042 }  
1043  
1044 /\*\*  
1045 \* If we're running on JDK 1.4 or later, initialize the cause for the given throwable.  
1046 \*  
1047 \* @param throwable The throwable.  
1048 \* @param cause The cause of the throwable.  
1049 \* @return true if the cause was initialized, otherwise false.  
1050 \* @since 1.8.0  
1051 \*/  
1052 public boolean initCause(final Throwable throwable, final Throwable cause) {  
1053 if (INIT\_CAUSE\_METHOD != null && cause != null) {  
1054 try {  
1055 INIT\_CAUSE\_METHOD.invoke(throwable, new Object[] { cause });  
1056 return true;  
1057 } catch (final Throwable e) {  
1058 return false; // can't initialize cause  
1059 }  
1060 }  
1061 return false;  
1062 }  
1063  
1064 /\*\*  
1065 \* <p>Convert the value to an object of the specified class (if  
1066 \* possible).</p>  
1067 \*  
1068 \* @param value Value to be converted (may be null)  
1069 \* @param type Class of the value to be converted to  
1070 \* @return The converted value  
1071 \*  
1072 \* @throws ConversionException if thrown by an underlying Converter  
1073 \* @since 1.8.0  
1074 \*/  
1075 protected Object convert(final Object value, final Class<?> type) {  
1076 final Converter converter = getConvertUtils().lookup(type);  
1077 if (converter != null) {  
1078 log.trace(" USING CONVERTER " + converter);  
1079 return converter.convert(type, value);  
1080 } else {  
1081 return value;  
1082 }  
1083 }  
1084  
1085 /\*\*  
1086 \* Performs a type conversion of a property value before it is copied to a target  
1087 \* bean. This method delegates to {@link #convert(Object, Class)}, but <b>null</b>  
1088 \* values are not converted. This causes <b>null</b> values to be copied verbatim.  
1089 \*  
1090 \* @param value the value to be converted and copied  
1091 \* @param type the target type of the conversion  
1092 \* @return the converted value  
1093 \*/  
1094 private Object convertForCopy(final Object value, final Class<?> type) {  
1095 return (value != null) ? convert(value, type) : value;  
1096 }  
1097  
1098 /\*\*  
1099 \* Returns a <code>Method<code> allowing access to  
1100 \* {@link Throwable#initCause(Throwable)} method of {@link Throwable},  
1101 \* or <code>null</code> if the method  
1102 \* does not exist.  
1103 \*  
1104 \* @return A <code>Method<code> for <code>Throwable.initCause</code>, or  
1105 \* <code>null</code> if unavailable.  
1106 \*/  
1107 private static Method getInitCauseMethod() {  
1108 try {  
1109 final Class<?>[] paramsClasses = new Class<?>[] { Throwable.class };  
1110 return Throwable.class.getMethod("initCause", paramsClasses);  
1111 } catch (final NoSuchMethodException e) {  
1112 final Log log = LogFactory.getLog(BeanUtils.class);  
1113 if (log.isWarnEnabled()) {  
1114 log.warn("Throwable does not have initCause() method in JDK 1.3");  
1115 }  
1116 return null;  
1117 } catch (final Throwable e) {  
1118 final Log log = LogFactory.getLog(BeanUtils.class);  
1119 if (log.isWarnEnabled()) {  
1120 log.warn("Error getting the Throwable initCause() method", e);  
1121 }  
1122 return null;  
1123 }  
1124 }  
1125  
1126 /\*\*  
1127 \* Determines the type of a {@code DynaProperty}. Here a special treatment  
1128 \* is needed for mapped properties.  
1129 \*  
1130 \* @param dynaProperty the property descriptor  
1131 \* @param value the value object to be set for this property  
1132 \* @return the type of this property  
1133 \*/  
1134 private static Class<?> dynaPropertyType(final DynaProperty dynaProperty,  
1135 final Object value) {  
1136 if (!dynaProperty.isMapped()) {  
1137 return dynaProperty.getType();  
1138 }  
1139 return (value == null) ? String.class : value.getClass();  
1140 }  
1141}