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 \*/  
017package org.apache.commons.beanutils;  
018  
019import java.io.Serializable;  
020import java.lang.reflect.Array;  
021import java.math.BigDecimal;  
022import java.math.BigInteger;  
023import java.util.ArrayList;  
024import java.util.Date;  
025import java.util.HashMap;  
026import java.util.List;  
027import java.util.Map;  
028  
029import org.apache.commons.logging.Log;  
030import org.apache.commons.logging.LogFactory;  
031  
032/\*\*  
033 \* <p>DynaBean which automatically adds properties to the <code>DynaClass</code>  
034 \* and provides <i>Lazy List</i> and <i>Lazy Map</i> features.</p>  
035 \*  
036 \* <p>DynaBeans deal with three types of properties - <i>simple</i>, <i>indexed</i> and <i>mapped</i> and  
037 \* have the following <code>get()</code> and <code>set()</code> methods for  
038 \* each of these types:</p>  
039 \* <ul>  
040 \* <li><i>Simple</i> property methods - <code>get(name)</code> and  
041 \* <code>set(name, value)</code></li>  
042 \* <li><i>Indexed</i> property methods - <code>get(name, index)</code> and  
043 \* <code>set(name, index, value)</code></li>  
044 \* <li><i>Mapped</i> property methods - <code>get(name, key)</code> and  
045 \* <code>set(name, key, value)</code></li>  
046 \* </ul>  
047 \*  
048 \* <p><b><u>Getting Property Values</u></b></p>  
049 \* <p>Calling any of the <code>get()</code> methods, for a property which  
050 \* doesn't exist, returns <code>null</code> in this implementation.</p>  
051 \*  
052 \* <p><b><u>Setting Simple Properties</u></b></p>  
053 \* <p>The <code>LazyDynaBean</code> will automatically add a property to the <code>DynaClass</code>  
054 \* if it doesn't exist when the <code>set(name, value)</code> method is called.</p>  
055 \*  
056 \* <code>DynaBean myBean = new LazyDynaBean();</code></br>  
057 \* <code>myBean.set("myProperty", "myValue");</code></br>  
058 \*  
059 \* <p><b><u>Setting Indexed Properties</u></b></p>  
060 \* <p>If the property <b>doesn't</b> exist, the <code>LazyDynaBean</code> will automatically add  
061 \* a property with an <code>ArrayList</code> type to the <code>DynaClass</code> when  
062 \* the <code>set(name, index, value)</code> method is called.  
063 \* It will also instantiate a new <code>ArrayList</code> and automatically <i>grow</i>  
064 \* the <code>List</code> so that it is big enough to accomodate the index being set.  
065 \* <code>ArrayList</code> is the default indexed property that LazyDynaBean uses but  
066 \* this can be easily changed by overriding the <code>defaultIndexedProperty(name)</code>  
067 \* method.</p>  
068 \*  
069 \* <code>DynaBean myBean = new LazyDynaBean();</code></br>  
070 \* <code>myBean.set("myIndexedProperty", 0, "myValue1");</code></br>  
071 \* <code>myBean.set("myIndexedProperty", 1, "myValue2");</code></br>  
072 \*  
073 \* <p>If the indexed property <b>does</b> exist in the <code>DynaClass</code> but is set to  
074 \* <code>null</code> in the <code>LazyDynaBean</code>, then it will instantiate a  
075 \* new <code>List</code> or <code>Array</code> as specified by the property's type  
076 \* in the <code>DynaClass</code> and automatically <i>grow</i> the <code>List</code>  
077 \* or <code>Array</code> so that it is big enough to accomodate the index being set.</p>  
078 \*  
079 \* <code>DynaBean myBean = new LazyDynaBean();</code></br>  
080 \* <code>MutableDynaClass myClass = (MutableDynaClass)myBean.getDynaClass();</code></br>  
081 \* <code>myClass.add("myIndexedProperty", int[].class);</code></br>  
082 \* <code>myBean.set("myIndexedProperty", 0, new Integer(10));</code></br>  
083 \* <code>myBean.set("myIndexedProperty", 1, new Integer(20));</code></br>  
084 \*  
085 \* <p><b><u>Setting Mapped Properties</u></b></p>  
086 \* <p>If the property <b>doesn't</b> exist, the <code>LazyDynaBean</code> will automatically add  
087 \* a property with a <code>HashMap</code> type to the <code>DynaClass</code> and  
088 \* instantiate a new <code>HashMap</code> in the DynaBean when the  
089 \* <code>set(name, key, value)</code> method is called. <code>HashMap</code> is the default  
090 \* mapped property that LazyDynaBean uses but this can be easily changed by overriding  
091 \* the <code>defaultMappedProperty(name)</code> method.</p>  
092 \*  
093 \* <code>DynaBean myBean = new LazyDynaBean();</code></br>  
094 \* <code>myBean.set("myMappedProperty", "myKey", "myValue");</code></br>  
095 \*  
096 \* <p>If the mapped property <b>does</b> exist in the <code>DynaClass</code> but is set to  
097 \* <code>null</code> in the <code>LazyDynaBean</code>, then it will instantiate a  
098 \* new <code>Map</code> as specified by the property's type in the <code>DynaClass</code>.</p>  
099 \*  
100 \* <code>DynaBean myBean = new LazyDynaBean();</code></br>  
101 \* <code>MutableDynaClass myClass = (MutableDynaClass)myBean.getDynaClass();</code></br>  
102 \* <code>myClass.add("myMappedProperty", TreeMap.class);</code></br>  
103 \* <code>myBean.set("myMappedProperty", "myKey", "myValue");</code></br>  
104 \*  
105 \* <p><b><u><i>Restricted</i> DynaClass</u></b></p>  
106 \* <p><code>MutableDynaClass</code> have a facility to <i>restrict</i> the <code>DynaClass</code>  
107 \* so that its properties cannot be modified. If the <code>MutableDynaClass</code> is  
108 \* restricted then calling any of the <code>set()</code> methods for a property which  
109 \* doesn't exist will result in a <code>IllegalArgumentException</code> being thrown.</p>  
110 \*  
111 \* @version $Id$  
112 \* @see LazyDynaClass  
113 \*/  
114public class LazyDynaBean implements DynaBean, Serializable {  
115  
116  
117 /\*\*  
118 \* Commons Logging  
119 \*/  
120 private transient Log logger = LogFactory.getLog(LazyDynaBean.class);  
121  
122 /\*\* BigInteger Zero \*/  
123 protected static final BigInteger BigInteger\_ZERO = new BigInteger("0");  
124 /\*\* BigDecimal Zero \*/  
125 protected static final BigDecimal BigDecimal\_ZERO = new BigDecimal("0");  
126 /\*\* Character Space \*/  
127 protected static final Character Character\_SPACE = new Character(' ');  
128 /\*\* Byte Zero \*/  
129 protected static final Byte Byte\_ZERO = new Byte((byte)0);  
130 /\*\* Short Zero \*/  
131 protected static final Short Short\_ZERO = new Short((short)0);  
132 /\*\* Integer Zero \*/  
133 protected static final Integer Integer\_ZERO = new Integer(0);  
134 /\*\* Long Zero \*/  
135 protected static final Long Long\_ZERO = new Long(0);  
136 /\*\* Float Zero \*/  
137 protected static final Float Float\_ZERO = new Float((byte)0);  
138 /\*\* Double Zero \*/  
139 protected static final Double Double\_ZERO = new Double((byte)0);  
140  
141 /\*\*  
142 \* The <code>MutableDynaClass</code> "base class" that this DynaBean  
143 \* is associated with.  
144 \*/  
145 protected Map<String, Object> values;  
146  
147 /\*\* Map decorator for this DynaBean \*/  
148 private transient Map<String, Object> mapDecorator;  
149  
150 /\*\*  
151 \* The <code>MutableDynaClass</code> "base class" that this DynaBean  
152 \* is associated with.  
153 \*/  
154 protected MutableDynaClass dynaClass;  
155  
156  
157 // ------------------- Constructors ----------------------------------  
158  
159 /\*\*  
160 \* Construct a new <code>LazyDynaBean</code> with a <code>LazyDynaClass</code> instance.  
161 \*/  
162 public LazyDynaBean() {  
163 this(new LazyDynaClass());  
164 }  
165  
166 /\*\*  
167 \* Construct a new <code>LazyDynaBean</code> with a <code>LazyDynaClass</code> instance.  
168 \*  
169 \* @param name Name of this DynaBean class  
170 \*/  
171 public LazyDynaBean(final String name) {  
172 this(new LazyDynaClass(name));  
173 }  
174  
175 /\*\*  
176 \* Construct a new <code>DynaBean</code> associated with the specified  
177 \* <code>DynaClass</code> instance - if its not a <code>MutableDynaClass</code>  
178 \* then a new <code>LazyDynaClass</code> is created and the properties copied.  
179 \*  
180 \* @param dynaClass The DynaClass we are associated with  
181 \*/  
182 public LazyDynaBean(final DynaClass dynaClass) {  
183  
184 values = newMap();  
185  
186 if (dynaClass instanceof MutableDynaClass) {  
187 this.dynaClass = (MutableDynaClass)dynaClass;  
188 } else {  
189 this.dynaClass = new LazyDynaClass(dynaClass.getName(), dynaClass.getDynaProperties());  
190 }  
191  
192 }  
193  
194  
195 // ------------------- Public Methods ----------------------------------  
196  
197 /\*\*  
198 \* Return a Map representation of this DynaBean.  
199 \* </p>  
200 \* This, for example, could be used in JSTL in the following way to access  
201 \* a DynaBean's <code>fooProperty</code>:  
202 \* <ul><li><code>${myDynaBean.<b>map</b>.fooProperty}</code></li></ul>  
203 \*  
204 \* @return a Map representation of this DynaBean  
205 \*/  
206 public Map<String, Object> getMap() {  
207 // cache the Map  
208 if (mapDecorator == null) {  
209 mapDecorator = new DynaBeanPropertyMapDecorator(this);  
210 }  
211 return mapDecorator;  
212 }  
213  
214 /\*\*  
215 \* <p>Return the size of an indexed or mapped property.</p>  
216 \*  
217 \* @param name Name of the property  
218 \* @return The indexed or mapped property size  
219 \* @throws IllegalArgumentException if no property name is specified  
220 \*/  
221 public int size(final String name) {  
222  
223 if (name == null) {  
224 throw new IllegalArgumentException("No property name specified");  
225 }  
226  
227 final Object value = values.get(name);  
228 if (value == null) {  
229 return 0;  
230 }  
231  
232 if (value instanceof Map) {  
233 return ((Map<?, ?>)value).size();  
234 }  
235  
236 if (value instanceof List) {  
237 return ((List<?>)value).size();  
238 }  
239  
240 if ((value.getClass().isArray())) {  
241 return Array.getLength(value);  
242 }  
243  
244 return 0;  
245  
246 }  
247  
248 // ------------------- DynaBean Methods ----------------------------------  
249  
250 /\*\*  
251 \* Does the specified mapped property contain a value for the specified  
252 \* key value?  
253 \*  
254 \* @param name Name of the property to check  
255 \* @param key Name of the key to check  
256 \* @return <code>true</code> if the mapped property contains a value for  
257 \* the specified key, otherwise <code>false</code>  
258 \*  
259 \* @throws IllegalArgumentException if no property name is specified  
260 \*/  
261 public boolean contains(final String name, final String key) {  
262  
263 if (name == null) {  
264 throw new IllegalArgumentException("No property name specified");  
265 }  
266  
267 final Object value = values.get(name);  
268 if (value == null) {  
269 return false;  
270 }  
271  
272 if (value instanceof Map) {  
273 return (((Map<?, ?>) value).containsKey(key));  
274 }  
275  
276 return false;  
277  
278 }  
279  
280 /\*\*  
281 \* <p>Return the value of a simple property with the specified name.</p>  
282 \*  
283 \* <p><strong>N.B.</strong> Returns <code>null</code> if there is no property  
284 \* of the specified name.</p>  
285 \*  
286 \* @param name Name of the property whose value is to be retrieved.  
287 \* @return The property's value  
288 \* @throws IllegalArgumentException if no property name is specified  
289 \*/  
290 public Object get(final String name) {  
291  
292 if (name == null) {  
293 throw new IllegalArgumentException("No property name specified");  
294 }  
295  
296 // Value found  
297 Object value = values.get(name);  
298 if (value != null) {  
299 return value;  
300 }  
301  
302 // Property doesn't exist  
303 if (!isDynaProperty(name)) {  
304 return null;  
305 }  
306  
307 // Property doesn't exist  
308 value = createProperty(name, dynaClass.getDynaProperty(name).getType());  
309  
310 if (value != null) {  
311 set(name, value);  
312 }  
313  
314 return value;  
315  
316 }  
317  
318 /\*\*  
319 \* <p>Return the value of an indexed property with the specified name.</p>  
320 \*  
321 \* <p><strong>N.B.</strong> Returns <code>null</code> if there is no 'indexed'  
322 \* property of the specified name.</p>  
323 \*  
324 \* @param name Name of the property whose value is to be retrieved  
325 \* @param index Index of the value to be retrieved  
326 \* @return The indexed property's value  
327 \*  
328 \* @throws IllegalArgumentException if the specified property  
329 \* exists, but is not indexed  
330 \* @throws IndexOutOfBoundsException if the specified index  
331 \* is outside the range of the underlying property  
332 \*/  
333 public Object get(final String name, final int index) {  
334  
335 // If its not a property, then create default indexed property  
336 if (!isDynaProperty(name)) {  
337 set(name, defaultIndexedProperty(name));  
338 }  
339  
340 // Get the indexed property  
341 Object indexedProperty = get(name);  
342  
343 // Check that the property is indexed  
344 if (!dynaClass.getDynaProperty(name).isIndexed()) {  
345 throw new IllegalArgumentException  
346 ("Non-indexed property for '" + name + "[" + index + "]' "  
347 + dynaClass.getDynaProperty(name).getName());  
348 }  
349  
350 // Grow indexed property to appropriate size  
351 indexedProperty = growIndexedProperty(name, indexedProperty, index);  
352  
353 // Return the indexed value  
354 if (indexedProperty.getClass().isArray()) {  
355 return Array.get(indexedProperty, index);  
356 } else if (indexedProperty instanceof List) {  
357 return ((List<?>)indexedProperty).get(index);  
358 } else {  
359 throw new IllegalArgumentException  
360 ("Non-indexed property for '" + name + "[" + index + "]' "  
361 + indexedProperty.getClass().getName());  
362 }  
363  
364 }  
365  
366 /\*\*  
367 \* <p>Return the value of a mapped property with the specified name.</p>  
368 \*  
369 \* <p><strong>N.B.</strong> Returns <code>null</code> if there is no 'mapped'  
370 \* property of the specified name.</p>  
371 \*  
372 \* @param name Name of the property whose value is to be retrieved  
373 \* @param key Key of the value to be retrieved  
374 \* @return The mapped property's value  
375 \*  
376 \* @throws IllegalArgumentException if the specified property  
377 \* exists, but is not mapped  
378 \*/  
379 public Object get(final String name, final String key) {  
380  
381 // If its not a property, then create default mapped property  
382 if (!isDynaProperty(name)) {  
383 set(name, defaultMappedProperty(name));  
384 }  
385  
386 // Get the mapped property  
387 final Object mappedProperty = get(name);  
388  
389 // Check that the property is mapped  
390 if (!dynaClass.getDynaProperty(name).isMapped()) {  
391 throw new IllegalArgumentException  
392 ("Non-mapped property for '" + name + "(" + key + ")' "  
393 + dynaClass.getDynaProperty(name).getType().getName());  
394 }  
395  
396 // Get the value from the Map  
397 if (mappedProperty instanceof Map) {  
398 return (((Map<?, ?>) mappedProperty).get(key));  
399 } else {  
400 throw new IllegalArgumentException  
401 ("Non-mapped property for '" + name + "(" + key + ")'"  
402 + mappedProperty.getClass().getName());  
403 }  
404  
405 }  
406  
407  
408 /\*\*  
409 \* Return the <code>DynaClass</code> instance that describes the set of  
410 \* properties available for this DynaBean.  
411 \*  
412 \* @return The associated DynaClass  
413 \*/  
414 public DynaClass getDynaClass() {  
415 return dynaClass;  
416 }  
417  
418 /\*\*  
419 \* Remove any existing value for the specified key on the  
420 \* specified mapped property.  
421 \*  
422 \* @param name Name of the property for which a value is to  
423 \* be removed  
424 \* @param key Key of the value to be removed  
425 \*  
426 \* @throws IllegalArgumentException if there is no property  
427 \* of the specified name  
428 \*/  
429 public void remove(final String name, final String key) {  
430  
431 if (name == null) {  
432 throw new IllegalArgumentException("No property name specified");  
433 }  
434  
435 final Object value = values.get(name);  
436 if (value == null) {  
437 return;  
438 }  
439  
440 if (value instanceof Map) {  
441 ((Map<?, ?>) value).remove(key);  
442 } else {  
443 throw new IllegalArgumentException  
444 ("Non-mapped property for '" + name + "(" + key + ")'"  
445 + value.getClass().getName());  
446 }  
447  
448 }  
449  
450 /\*\*  
451 \* Set the value of a simple property with the specified name.  
452 \*  
453 \* @param name Name of the property whose value is to be set  
454 \* @param value Value to which this property is to be set  
455 \*  
456 \* @throws IllegalArgumentException if this is not an existing property  
457 \* name for our DynaClass and the MutableDynaClass is restricted  
458 \* @throws ConversionException if the specified value cannot be  
459 \* converted to the type required for this property  
460 \* @throws NullPointerException if an attempt is made to set a  
461 \* primitive property to null  
462 \*/  
463 public void set(final String name, final Object value) {  
464  
465 // If the property doesn't exist, then add it  
466 if (!isDynaProperty(name)) {  
467  
468 if (dynaClass.isRestricted()) {  
469 throw new IllegalArgumentException  
470 ("Invalid property name '" + name + "' (DynaClass is restricted)");  
471 }  
472 if (value == null) {  
473 dynaClass.add(name);  
474 } else {  
475 dynaClass.add(name, value.getClass());  
476 }  
477  
478 }  
479  
480 final DynaProperty descriptor = dynaClass.getDynaProperty(name);  
481  
482 if (value == null) {  
483 if (descriptor.getType().isPrimitive()) {  
484 throw new NullPointerException  
485 ("Primitive value for '" + name + "'");  
486 }  
487 } else if (!isAssignable(descriptor.getType(), value.getClass())) {  
488 throw new ConversionException  
489 ("Cannot assign value of type '" +  
490 value.getClass().getName() +  
491 "' to property '" + name + "' of type '" +  
492 descriptor.getType().getName() + "'");  
493 }  
494  
495 // Set the property's value  
496 values.put(name, value);  
497  
498 }  
499  
500 /\*\*  
501 \* Set the value of an indexed property with the specified name.  
502 \*  
503 \* @param name Name of the property whose value is to be set  
504 \* @param index Index of the property to be set  
505 \* @param value Value to which this property is to be set  
506 \*  
507 \* @throws ConversionException if the specified value cannot be  
508 \* converted to the type required for this property  
509 \* @throws IllegalArgumentException if there is no property  
510 \* of the specified name  
511 \* @throws IllegalArgumentException if the specified property  
512 \* exists, but is not indexed  
513 \* @throws IndexOutOfBoundsException if the specified index  
514 \* is outside the range of the underlying property  
515 \*/  
516 public void set(final String name, final int index, final Object value) {  
517  
518 // If its not a property, then create default indexed property  
519 if (!isDynaProperty(name)) {  
520 set(name, defaultIndexedProperty(name));  
521 }  
522  
523 // Get the indexed property  
524 Object indexedProperty = get(name);  
525  
526 // Check that the property is indexed  
527 if (!dynaClass.getDynaProperty(name).isIndexed()) {  
528 throw new IllegalArgumentException  
529 ("Non-indexed property for '" + name + "[" + index + "]'"  
530 + dynaClass.getDynaProperty(name).getType().getName());  
531 }  
532  
533 // Grow indexed property to appropriate size  
534 indexedProperty = growIndexedProperty(name, indexedProperty, index);  
535  
536 // Set the value in an array  
537 if (indexedProperty.getClass().isArray()) {  
538 Array.set(indexedProperty, index, value);  
539 } else if (indexedProperty instanceof List) {  
540 @SuppressWarnings("unchecked")  
541 final  
542 // Indexed properties are stored in a List<Object>  
543 List<Object> values = (List<Object>) indexedProperty;  
544 values.set(index, value);  
545 } else {  
546 throw new IllegalArgumentException  
547 ("Non-indexed property for '" + name + "[" + index + "]' "  
548 + indexedProperty.getClass().getName());  
549 }  
550  
551 }  
552  
553 /\*\*  
554 \* Set the value of a mapped property with the specified name.  
555 \*  
556 \* @param name Name of the property whose value is to be set  
557 \* @param key Key of the property to be set  
558 \* @param value Value to which this property is to be set  
559 \*  
560 \* @throws ConversionException if the specified value cannot be  
561 \* converted to the type required for this property  
562 \* @throws IllegalArgumentException if there is no property  
563 \* of the specified name  
564 \* @throws IllegalArgumentException if the specified property  
565 \* exists, but is not mapped  
566 \*/  
567 public void set(final String name, final String key, final Object value) {  
568  
569 // If the 'mapped' property doesn't exist, then add it  
570 if (!isDynaProperty(name)) {  
571 set(name, defaultMappedProperty(name));  
572 }  
573  
574 // Get the mapped property  
575 final Object mappedProperty = get(name);  
576  
577 // Check that the property is mapped  
578 if (!dynaClass.getDynaProperty(name).isMapped()) {  
579 throw new IllegalArgumentException  
580 ("Non-mapped property for '" + name + "(" + key + ")'"  
581 + dynaClass.getDynaProperty(name).getType().getName());  
582 }  
583  
584 // Set the value in the Map  
585 @SuppressWarnings("unchecked")  
586 final  
587 // mapped properties are stored in a Map<String, Object>  
588 Map<String, Object> valuesMap = (Map<String, Object>) mappedProperty;  
589 valuesMap.put(key, value);  
590  
591 }  
592  
593 // ------------------- protected Methods ----------------------------------  
594  
595 /\*\*  
596 \* Grow the size of an indexed property  
597 \* @param name The name of the property  
598 \* @param indexedProperty The current property value  
599 \* @param index The indexed value to grow the property to (i.e. one less than  
600 \* the required size)  
601 \* @return The new property value (grown to the appropriate size)  
602 \*/  
603 protected Object growIndexedProperty(final String name, Object indexedProperty, final int index) {  
604  
605 // Grow a List to the appropriate size  
606 if (indexedProperty instanceof List) {  
607  
608 @SuppressWarnings("unchecked")  
609 final  
610 // Indexed properties are stored as List<Object>  
611 List<Object> list = (List<Object>)indexedProperty;  
612 while (index >= list.size()) {  
613 final Class<?> contentType = getDynaClass().getDynaProperty(name).getContentType();  
614 Object value = null;  
615 if (contentType != null) {  
616 value = createProperty(name+"["+list.size()+"]", contentType);  
617 }  
618 list.add(value);  
619 }  
620  
621 }  
622  
623 // Grow an Array to the appropriate size  
624 if ((indexedProperty.getClass().isArray())) {  
625  
626 final int length = Array.getLength(indexedProperty);  
627 if (index >= length) {  
628 final Class<?> componentType = indexedProperty.getClass().getComponentType();  
629 final Object newArray = Array.newInstance(componentType, (index + 1));  
630 System.arraycopy(indexedProperty, 0, newArray, 0, length);  
631 indexedProperty = newArray;  
632 set(name, indexedProperty);  
633 final int newLength = Array.getLength(indexedProperty);  
634 for (int i = length; i < newLength; i++) {  
635 Array.set(indexedProperty, i, createProperty(name+"["+i+"]", componentType));  
636 }  
637 }  
638 }  
639  
640 return indexedProperty;  
641  
642 }  
643  
644 /\*\*  
645 \* Create a new Instance of a Property  
646 \* @param name The name of the property  
647 \* @param type The class of the property  
648 \* @return The new value  
649 \*/  
650 protected Object createProperty(final String name, final Class<?> type) {  
651 if (type == null) {  
652 return null;  
653 }  
654  
655 // Create Lists, arrays or DynaBeans  
656 if (type.isArray() || List.class.isAssignableFrom(type)) {  
657 return createIndexedProperty(name, type);  
658 }  
659  
660 if (Map.class.isAssignableFrom(type)) {  
661 return createMappedProperty(name, type);  
662 }  
663  
664 if (DynaBean.class.isAssignableFrom(type)) {  
665 return createDynaBeanProperty(name, type);  
666 }  
667  
668 if (type.isPrimitive()) {  
669 return createPrimitiveProperty(name, type);  
670 }  
671  
672 if (Number.class.isAssignableFrom(type)) {  
673 return createNumberProperty(name, type);  
674 }  
675  
676 return createOtherProperty(name, type);  
677  
678 }  
679  
680 /\*\*  
681 \* Create a new Instance of an 'Indexed' Property  
682 \* @param name The name of the property  
683 \* @param type The class of the property  
684 \* @return The new value  
685 \*/  
686 protected Object createIndexedProperty(final String name, final Class<?> type) {  
687  
688 // Create the indexed object  
689 Object indexedProperty = null;  
690  
691 if (type == null) {  
692  
693 indexedProperty = defaultIndexedProperty(name);  
694  
695 } else if (type.isArray()) {  
696  
697 indexedProperty = Array.newInstance(type.getComponentType(), 0);  
698  
699 } else if (List.class.isAssignableFrom(type)) {  
700 if (type.isInterface()) {  
701 indexedProperty = defaultIndexedProperty(name);  
702 } else {  
703 try {  
704 indexedProperty = type.newInstance();  
705 }  
706 catch (final Exception ex) {  
707 throw new IllegalArgumentException  
708 ("Error instantiating indexed property of type '" +  
709 type.getName() + "' for '" + name + "' " + ex);  
710 }  
711 }  
712 } else {  
713  
714 throw new IllegalArgumentException  
715 ("Non-indexed property of type '" + type.getName() + "' for '" + name + "'");  
716 }  
717  
718 return indexedProperty;  
719  
720 }  
721  
722 /\*\*  
723 \* Create a new Instance of a 'Mapped' Property  
724 \* @param name The name of the property  
725 \* @param type The class of the property  
726 \* @return The new value  
727 \*/  
728 protected Object createMappedProperty(final String name, final Class<?> type) {  
729  
730 // Create the mapped object  
731 Object mappedProperty = null;  
732  
733 if (type == null) {  
734  
735 mappedProperty = defaultMappedProperty(name);  
736  
737 } else if (type.isInterface()) {  
738  
739 mappedProperty = defaultMappedProperty(name);  
740  
741 } else if (Map.class.isAssignableFrom(type)) {  
742 try {  
743 mappedProperty = type.newInstance();  
744 }  
745 catch (final Exception ex) {  
746 throw new IllegalArgumentException  
747 ("Error instantiating mapped property of type '" +  
748 type.getName() + "' for '" + name + "' " + ex);  
749 }  
750 } else {  
751  
752 throw new IllegalArgumentException  
753 ("Non-mapped property of type '" + type.getName() + "' for '" + name + "'");  
754 }  
755  
756 return mappedProperty;  
757  
758 }  
759  
760 /\*\*  
761 \* Create a new Instance of a 'DynaBean' Property.  
762 \* @param name The name of the property  
763 \* @param type The class of the property  
764 \* @return The new value  
765 \*/  
766 protected Object createDynaBeanProperty(final String name, final Class<?> type) {  
767 try {  
768 return type.newInstance();  
769 }  
770 catch (final Exception ex) {  
771 if (logger().isWarnEnabled()) {  
772 logger().warn("Error instantiating DynaBean property of type '" +  
773 type.getName() + "' for '" + name + "' " + ex);  
774 }  
775 return null;  
776 }  
777 }  
778  
779 /\*\*  
780 \* Create a new Instance of a 'Primitive' Property.  
781 \* @param name The name of the property  
782 \* @param type The class of the property  
783 \* @return The new value  
784 \*/  
785 protected Object createPrimitiveProperty(final String name, final Class<?> type) {  
786  
787 if (type == Boolean.TYPE) {  
788 return Boolean.FALSE;  
789 } else if (type == Integer.TYPE) {  
790 return Integer\_ZERO;  
791 } else if (type == Long.TYPE) {  
792 return Long\_ZERO;  
793 } else if (type == Double.TYPE) {  
794 return Double\_ZERO;  
795 } else if (type == Float.TYPE) {  
796 return Float\_ZERO;  
797 } else if (type == Byte.TYPE) {  
798 return Byte\_ZERO;  
799 } else if (type == Short.TYPE) {  
800 return Short\_ZERO;  
801 } else if (type == Character.TYPE) {  
802 return Character\_SPACE;  
803 } else {  
804 return null;  
805 }  
806  
807 }  
808  
809 /\*\*  
810 \* Create a new Instance of a <code>java.lang.Number</code> Property.  
811 \* @param name The name of the property  
812 \* @param type The class of the property  
813 \* @return The new value  
814 \*/  
815 protected Object createNumberProperty(final String name, final Class<?> type) {  
816  
817 return null;  
818  
819 }  
820  
821 /\*\*  
822 \* Create a new Instance of other Property types  
823 \* @param name The name of the property  
824 \* @param type The class of the property  
825 \* @return The new value  
826 \*/  
827 protected Object createOtherProperty(final String name, final Class<?> type) {  
828  
829 if (type == Object.class ||  
830 type == String.class ||  
831 type == Boolean.class ||  
832 type == Character.class ||  
833 Date.class.isAssignableFrom(type)) {  
834  
835 return null;  
836  
837 }  
838  
839 try {  
840 return type.newInstance();  
841 }  
842 catch (final Exception ex) {  
843 if (logger().isWarnEnabled()) {  
844 logger().warn("Error instantiating property of type '" + type.getName() + "' for '" + name + "' " + ex);  
845 }  
846 return null;  
847 }  
848 }  
849  
850 /\*\*  
851 \* <p>Creates a new <code>ArrayList</code> for an 'indexed' property  
852 \* which doesn't exist.</p>  
853 \*  
854 \* <p>This method should be overridden if an alternative <code>List</code>  
855 \* or <code>Array</code> implementation is required for 'indexed' properties.</p>  
856 \*  
857 \* @param name Name of the 'indexed property.  
858 \* @return The default value for an indexed property (java.util.ArrayList)  
859 \*/  
860 protected Object defaultIndexedProperty(final String name) {  
861 return new ArrayList<Object>();  
862 }  
863  
864 /\*\*  
865 \* <p>Creates a new <code>HashMap</code> for a 'mapped' property  
866 \* which doesn't exist.</p>  
867 \*  
868 \* <p>This method can be overridden if an alternative <code>Map</code>  
869 \* implementation is required for 'mapped' properties.</p>  
870 \*  
871 \* @param name Name of the 'mapped property.  
872 \* @return The default value for a mapped property (java.util.HashMap)  
873 \*/  
874 protected Map<String, Object> defaultMappedProperty(final String name) {  
875 return new HashMap<String, Object>();  
876 }  
877  
878 /\*\*  
879 \* Indicates if there is a property with the specified name.  
880 \* @param name The name of the property to check  
881 \* @return <code>true</code> if there is a property of the  
882 \* specified name, otherwise <code>false</code>  
883 \*/  
884 protected boolean isDynaProperty(final String name) {  
885  
886 if (name == null) {  
887 throw new IllegalArgumentException("No property name specified");  
888 }  
889  
890 // Handle LazyDynaClasses  
891 if (dynaClass instanceof LazyDynaClass) {  
892 return ((LazyDynaClass)dynaClass).isDynaProperty(name);  
893 }  
894  
895 // Handle other MutableDynaClass  
896 return dynaClass.getDynaProperty(name) == null ? false : true;  
897  
898 }  
899  
900 /\*\*  
901 \* Is an object of the source class assignable to the destination class?  
902 \*  
903 \* @param dest Destination class  
904 \* @param source Source class  
905 \* @return <code>true</code> if the source class is assignable to the  
906 \* destination class, otherwise <code>false</code>  
907 \*/  
908 protected boolean isAssignable(final Class<?> dest, final Class<?> source) {  
909  
910 if (dest.isAssignableFrom(source) ||  
911 ((dest == Boolean.TYPE) && (source == Boolean.class)) ||  
912 ((dest == Byte.TYPE) && (source == Byte.class)) ||  
913 ((dest == Character.TYPE) && (source == Character.class)) ||  
914 ((dest == Double.TYPE) && (source == Double.class)) ||  
915 ((dest == Float.TYPE) && (source == Float.class)) ||  
916 ((dest == Integer.TYPE) && (source == Integer.class)) ||  
917 ((dest == Long.TYPE) && (source == Long.class)) ||  
918 ((dest == Short.TYPE) && (source == Short.class))) {  
919 return (true);  
920 } else {  
921 return (false);  
922 }  
923  
924 }  
925  
926 /\*\*  
927 \* <p>Creates a new instance of the <code>Map</code>.</p>  
928 \* @return a new Map instance  
929 \*/  
930 protected Map<String, Object> newMap() {  
931 return new HashMap<String, Object>();  
932 }  
933  
934 /\*\*  
935 \* <p>Returns the <code>Log</code>.  
936 \*/  
937 private Log logger() {  
938 if (logger == null) {  
939 logger = LogFactory.getLog(LazyDynaBean.class);  
940 }  
941 return logger;  
942 }  
943  
944}