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.io.Serializable;  
023import java.lang.reflect.Array;  
024import java.util.HashMap;  
025import java.util.List;  
026import java.util.Map;  
027  
028  
029/\*\*  
030 \* <p>Minimal implementation of the <code>DynaBean</code> interface. Can be  
031 \* used as a convenience base class for more sophisticated implementations.</p>  
032 \*  
033 \* <p><strong>IMPLEMENTATION NOTE</strong> - Instances of this class that are  
034 \* accessed from multiple threads simultaneously need to be synchronized.</p>  
035 \*  
036 \* <p><strong>IMPLEMENTATION NOTE</strong> - Instances of this class can be  
037 \* successfully serialized and deserialized <strong>ONLY</strong> if all  
038 \* property values are <code>Serializable</code>.</p>  
039 \*  
040 \* @version $Id$  
041 \*/  
042  
043public class BasicDynaBean implements DynaBean, Serializable {  
044  
045  
046 // ---------------------------------------------------------- Constructors  
047  
048  
049 /\*\*  
050 \* Construct a new <code>DynaBean</code> associated with the specified  
051 \* <code>DynaClass</code> instance.  
052 \*  
053 \* @param dynaClass The DynaClass we are associated with  
054 \*/  
055 public BasicDynaBean(final DynaClass dynaClass) {  
056  
057 super();  
058 this.dynaClass = dynaClass;  
059  
060 }  
061  
062  
063 // ---------------------------------------------------- Instance Variables  
064  
065  
066 /\*\*  
067 \* The <code>DynaClass</code> "base class" that this DynaBean  
068 \* is associated with.  
069 \*/  
070 protected DynaClass dynaClass = null;  
071  
072  
073 /\*\*  
074 \* The set of property values for this DynaBean, keyed by property name.  
075 \*/  
076 protected HashMap<String, Object> values = new HashMap<String, Object>();  
077  
078 /\*\* Map decorator for this DynaBean \*/  
079 private transient Map<String, Object> mapDecorator;  
080  
081 /\*\*  
082 \* Return a Map representation of this DynaBean.  
083 \* </p>  
084 \* This, for example, could be used in JSTL in the following way to access  
085 \* a DynaBean's <code>fooProperty</code>:  
086 \* <ul><li><code>${myDynaBean.<b>map</b>.fooProperty}</code></li></ul>  
087 \*  
088 \* @return a Map representation of this DynaBean  
089 \* @since 1.8.0  
090 \*/  
091 public Map<String, Object> getMap() {  
092  
093 // cache the Map  
094 if (mapDecorator == null) {  
095 mapDecorator = new DynaBeanPropertyMapDecorator(this);  
096 }  
097 return mapDecorator;  
098  
099 }  
100  
101 // ------------------------------------------------------ DynaBean Methods  
102  
103  
104 /\*\*  
105 \* Does the specified mapped property contain a value for the specified  
106 \* key value?  
107 \*  
108 \* @param name Name of the property to check  
109 \* @param key Name of the key to check  
110 \* @return <code>true</code> if the mapped property contains a value for  
111 \* the specified key, otherwise <code>false</code>  
112 \*  
113 \* @throws IllegalArgumentException if there is no property  
114 \* of the specified name  
115 \*/  
116 public boolean contains(final String name, final String key) {  
117  
118 final Object value = values.get(name);  
119 if (value == null) {  
120 throw new NullPointerException  
121 ("No mapped value for '" + name + "(" + key + ")'");  
122 } else if (value instanceof Map) {  
123 return (((Map<?, ?>) value).containsKey(key));  
124 } else {  
125 throw new IllegalArgumentException  
126 ("Non-mapped property for '" + name + "(" + key + ")'");  
127 }  
128  
129 }  
130  
131  
132 /\*\*  
133 \* Return the value of a simple property with the specified name.  
134 \*  
135 \* @param name Name of the property whose value is to be retrieved  
136 \* @return The property's value  
137 \*  
138 \* @throws IllegalArgumentException if there is no property  
139 \* of the specified name  
140 \*/  
141 public Object get(final String name) {  
142  
143 // Return any non-null value for the specified property  
144 final Object value = values.get(name);  
145 if (value != null) {  
146 return (value);  
147 }  
148  
149 // Return a null value for a non-primitive property  
150 final Class<?> type = getDynaProperty(name).getType();  
151 if (!type.isPrimitive()) {  
152 return (value);  
153 }  
154  
155 // Manufacture default values for primitive properties  
156 if (type == Boolean.TYPE) {  
157 return (Boolean.FALSE);  
158 } else if (type == Byte.TYPE) {  
159 return (new Byte((byte) 0));  
160 } else if (type == Character.TYPE) {  
161 return (new Character((char) 0));  
162 } else if (type == Double.TYPE) {  
163 return (new Double(0.0));  
164 } else if (type == Float.TYPE) {  
165 return (new Float((float) 0.0));  
166 } else if (type == Integer.TYPE) {  
167 return (new Integer(0));  
168 } else if (type == Long.TYPE) {  
169 return (new Long(0));  
170 } else if (type == Short.TYPE) {  
171 return (new Short((short) 0));  
172 } else {  
173 return (null);  
174 }  
175  
176 }  
177  
178  
179 /\*\*  
180 \* Return the value of an indexed property with the specified name.  
181 \*  
182 \* @param name Name of the property whose value is to be retrieved  
183 \* @param index Index of the value to be retrieved  
184 \* @return The indexed property's value  
185 \*  
186 \* @throws IllegalArgumentException if there is no property  
187 \* of the specified name  
188 \* @throws IllegalArgumentException if the specified property  
189 \* exists, but is not indexed  
190 \* @throws IndexOutOfBoundsException if the specified index  
191 \* is outside the range of the underlying property  
192 \* @throws NullPointerException if no array or List has been  
193 \* initialized for this property  
194 \*/  
195 public Object get(final String name, final int index) {  
196  
197 final Object value = values.get(name);  
198 if (value == null) {  
199 throw new NullPointerException  
200 ("No indexed value for '" + name + "[" + index + "]'");  
201 } else if (value.getClass().isArray()) {  
202 return (Array.get(value, index));  
203 } else if (value instanceof List) {  
204 return ((List<?>) value).get(index);  
205 } else {  
206 throw new IllegalArgumentException  
207 ("Non-indexed property for '" + name + "[" + index + "]'");  
208 }  
209  
210 }  
211  
212  
213 /\*\*  
214 \* Return the value of a mapped property with the specified name,  
215 \* or <code>null</code> if there is no value for the specified key.  
216 \*  
217 \* @param name Name of the property whose value is to be retrieved  
218 \* @param key Key of the value to be retrieved  
219 \* @return The mapped property's value  
220 \*  
221 \* @throws IllegalArgumentException if there is no property  
222 \* of the specified name  
223 \* @throws IllegalArgumentException if the specified property  
224 \* exists, but is not mapped  
225 \*/  
226 public Object get(final String name, final String key) {  
227  
228 final Object value = values.get(name);  
229 if (value == null) {  
230 throw new NullPointerException  
231 ("No mapped value for '" + name + "(" + key + ")'");  
232 } else if (value instanceof Map) {  
233 return (((Map<?, ?>) value).get(key));  
234 } else {  
235 throw new IllegalArgumentException  
236 ("Non-mapped property for '" + name + "(" + key + ")'");  
237 }  
238  
239 }  
240  
241  
242 /\*\*  
243 \* Return the <code>DynaClass</code> instance that describes the set of  
244 \* properties available for this DynaBean.  
245 \*  
246 \* @return The associated DynaClass  
247 \*/  
248 public DynaClass getDynaClass() {  
249  
250 return (this.dynaClass);  
251  
252 }  
253  
254  
255 /\*\*  
256 \* Remove any existing value for the specified key on the  
257 \* specified mapped property.  
258 \*  
259 \* @param name Name of the property for which a value is to  
260 \* be removed  
261 \* @param key Key of the value to be removed  
262 \*  
263 \* @throws IllegalArgumentException if there is no property  
264 \* of the specified name  
265 \*/  
266 public void remove(final String name, final String key) {  
267  
268 final Object value = values.get(name);  
269 if (value == null) {  
270 throw new NullPointerException  
271 ("No mapped value for '" + name + "(" + key + ")'");  
272 } else if (value instanceof Map) {  
273 ((Map<?, ?>) value).remove(key);  
274 } else {  
275 throw new IllegalArgumentException  
276 ("Non-mapped property for '" + name + "(" + key + ")'");  
277 }  
278  
279 }  
280  
281  
282 /\*\*  
283 \* Set the value of a simple property with the specified name.  
284 \*  
285 \* @param name Name of the property whose value is to be set  
286 \* @param value Value to which this property is to be set  
287 \*  
288 \* @throws ConversionException if the specified value cannot be  
289 \* converted to the type required for this property  
290 \* @throws IllegalArgumentException if there is no property  
291 \* of the specified name  
292 \* @throws NullPointerException if an attempt is made to set a  
293 \* primitive property to null  
294 \*/  
295 public void set(final String name, final Object value) {  
296  
297 final DynaProperty descriptor = getDynaProperty(name);  
298 if (value == null) {  
299 if (descriptor.getType().isPrimitive()) {  
300 throw new NullPointerException  
301 ("Primitive value for '" + name + "'");  
302 }  
303 } else if (!isAssignable(descriptor.getType(), value.getClass())) {  
304 throw new ConversionException  
305 ("Cannot assign value of type '" +  
306 value.getClass().getName() +  
307 "' to property '" + name + "' of type '" +  
308 descriptor.getType().getName() + "'");  
309 }  
310 values.put(name, value);  
311  
312 }  
313  
314  
315 /\*\*  
316 \* Set the value of an indexed property with the specified name.  
317 \*  
318 \* @param name Name of the property whose value is to be set  
319 \* @param index Index of the property to be set  
320 \* @param value Value to which this property is to be set  
321 \*  
322 \* @throws ConversionException if the specified value cannot be  
323 \* converted to the type required for this property  
324 \* @throws IllegalArgumentException if there is no property  
325 \* of the specified name  
326 \* @throws IllegalArgumentException if the specified property  
327 \* exists, but is not indexed  
328 \* @throws IndexOutOfBoundsException if the specified index  
329 \* is outside the range of the underlying property  
330 \*/  
331 public void set(final String name, final int index, final Object value) {  
332  
333 final Object prop = values.get(name);  
334 if (prop == null) {  
335 throw new NullPointerException  
336 ("No indexed value for '" + name + "[" + index + "]'");  
337 } else if (prop.getClass().isArray()) {  
338 Array.set(prop, index, value);  
339 } else if (prop instanceof List) {  
340 try {  
341 @SuppressWarnings("unchecked")  
342 final  
343 // This is safe to cast because list properties are always  
344 // of type Object  
345 List<Object> list = (List<Object>) prop;  
346 list.set(index, value);  
347 } catch (final ClassCastException e) {  
348 throw new ConversionException(e.getMessage());  
349 }  
350 } else {  
351 throw new IllegalArgumentException  
352 ("Non-indexed property for '" + name + "[" + index + "]'");  
353 }  
354  
355 }  
356  
357  
358 /\*\*  
359 \* Set the value of a mapped property with the specified name.  
360 \*  
361 \* @param name Name of the property whose value is to be set  
362 \* @param key Key of the property to be set  
363 \* @param value Value to which this property is to be set  
364 \*  
365 \* @throws ConversionException if the specified value cannot be  
366 \* converted to the type required for this property  
367 \* @throws IllegalArgumentException if there is no property  
368 \* of the specified name  
369 \* @throws IllegalArgumentException if the specified property  
370 \* exists, but is not mapped  
371 \*/  
372 public void set(final String name, final String key, final Object value) {  
373  
374 final Object prop = values.get(name);  
375 if (prop == null) {  
376 throw new NullPointerException  
377 ("No mapped value for '" + name + "(" + key + ")'");  
378 } else if (prop instanceof Map) {  
379 @SuppressWarnings("unchecked")  
380 final  
381 // This is safe to cast because mapped properties are always  
382 // maps of types String -> Object  
383 Map<String, Object> map = (Map<String, Object>) prop;  
384 map.put(key, value);  
385 } else {  
386 throw new IllegalArgumentException  
387 ("Non-mapped property for '" + name + "(" + key + ")'");  
388 }  
389  
390 }  
391  
392  
393 // ------------------------------------------------------ Protected Methods  
394  
395  
396 /\*\*  
397 \* Return the property descriptor for the specified property name.  
398 \*  
399 \* @param name Name of the property for which to retrieve the descriptor  
400 \* @return The property descriptor  
401 \*  
402 \* @throws IllegalArgumentException if this is not a valid property  
403 \* name for our DynaClass  
404 \*/  
405 protected DynaProperty getDynaProperty(final String name) {  
406  
407 final DynaProperty descriptor = getDynaClass().getDynaProperty(name);  
408 if (descriptor == null) {  
409 throw new IllegalArgumentException  
410 ("Invalid property name '" + name + "'");  
411 }  
412 return (descriptor);  
413  
414 }  
415  
416  
417 /\*\*  
418 \* Is an object of the source class assignable to the destination class?  
419 \*  
420 \* @param dest Destination class  
421 \* @param source Source class  
422 \* @return <code>true</code> if the source class is assignable to the  
423 \* destination class, otherwise <code>false</code>  
424 \*/  
425 protected boolean isAssignable(final Class<?> dest, final Class<?> source) {  
426  
427 if (dest.isAssignableFrom(source) ||  
428 ((dest == Boolean.TYPE) && (source == Boolean.class)) ||  
429 ((dest == Byte.TYPE) && (source == Byte.class)) ||  
430 ((dest == Character.TYPE) && (source == Character.class)) ||  
431 ((dest == Double.TYPE) && (source == Double.class)) ||  
432 ((dest == Float.TYPE) && (source == Float.class)) ||  
433 ((dest == Integer.TYPE) && (source == Integer.class)) ||  
434 ((dest == Long.TYPE) && (source == Long.class)) ||  
435 ((dest == Short.TYPE) && (source == Short.class))) {  
436 return (true);  
437 } else {  
438 return (false);  
439 }  
440  
441 }  
442  
443  
444}