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.util.ArrayList;  
020import java.util.Collection;  
021import java.util.Collections;  
022import java.util.HashSet;  
023import java.util.List;  
024import java.util.Map;  
025import java.util.Set;  
026  
027/\*\*  
028 \* <p>A base class for decorators providing <code>Map</code> behavior on  
029 \* {@link DynaBean}s.</p>  
030 \*  
031 \* <p>The motivation for this implementation is to provide access to {@link DynaBean}  
032 \* properties in technologies that are unaware of BeanUtils and {@link DynaBean}s -  
033 \* such as the expression languages of JSTL and JSF.</p>  
034 \*  
035 \* <p>This rather technical base class implements the methods of the  
036 \* {@code Map} interface on top of a {@code DynaBean}. It was introduced  
037 \* to handle generic parameters in a meaningful way without breaking  
038 \* backwards compatibility of the {@link DynaBeanMapDecorator} class: A  
039 \* map wrapping a {@code DynaBean} should be of type {@code Map<String, Object>}.  
040 \* However, when using these generic parameters in {@code DynaBeanMapDecorator}  
041 \* this would be an incompatible change (as method signatures would have to  
042 \* be adapted). To solve this problem, this generic base class is added  
043 \* which allows specifying the key type as parameter. This makes it easy to  
044 \* have a new subclass using the correct generic parameters while  
045 \* {@code DynaBeanMapDecorator} could still remain with compatible  
046 \* parameters.</p>  
047 \*  
048 \* @param <K> the type of the keys in the decorated map  
049 \* @since BeanUtils 1.9.0  
050 \* @version $Id$  
051 \*/  
052public abstract class BaseDynaBeanMapDecorator<K> implements Map<K, Object> {  
053  
054 private final DynaBean dynaBean;  
055 private final boolean readOnly;  
056 private transient Set<K> keySet;  
057  
058 // ------------------- Constructors ----------------------------------  
059  
060 /\*\*  
061 \* Constructs a read only Map for the specified  
062 \* {@link DynaBean}.  
063 \*  
064 \* @param dynaBean The dyna bean being decorated  
065 \* @throws IllegalArgumentException if the {@link DynaBean} is null.  
066 \*/  
067 public BaseDynaBeanMapDecorator(final DynaBean dynaBean) {  
068 this(dynaBean, true);  
069 }  
070  
071 /\*\*  
072 \* Construct a Map for the specified {@link DynaBean}.  
073 \*  
074 \* @param dynaBean The dyna bean being decorated  
075 \* @param readOnly <code>true</code> if the Map is read only  
076 \* otherwise <code>false</code>  
077 \* @throws IllegalArgumentException if the {@link DynaBean} is null.  
078 \*/  
079 public BaseDynaBeanMapDecorator(final DynaBean dynaBean, final boolean readOnly) {  
080 if (dynaBean == null) {  
081 throw new IllegalArgumentException("DynaBean is null");  
082 }  
083 this.dynaBean = dynaBean;  
084 this.readOnly = readOnly;  
085 }  
086  
087  
088 // ------------------- public Methods --------------------------------  
089  
090  
091 /\*\*  
092 \* Indicate whether the Map is read only.  
093 \*  
094 \* @return <code>true</code> if the Map is read only,  
095 \* otherwise <code>false</code>.  
096 \*/  
097 public boolean isReadOnly() {  
098 return readOnly;  
099 }  
100  
101 // ------------------- java.util.Map Methods -------------------------  
102  
103 /\*\*  
104 \* clear() operation is not supported.  
105 \*  
106 \* @throws UnsupportedOperationException  
107 \*/  
108 public void clear() {  
109 throw new UnsupportedOperationException();  
110 }  
111  
112 /\*\*  
113 \* Indicate whether the {@link DynaBean} contains a specified  
114 \* value for one (or more) of its properties.  
115 \*  
116 \* @param key The {@link DynaBean}'s property name  
117 \* @return <code>true</code> if one of the {@link DynaBean}'s  
118 \* properties contains a specified value.  
119 \*/  
120 public boolean containsKey(final Object key) {  
121 final DynaClass dynaClass = getDynaBean().getDynaClass();  
122 final DynaProperty dynaProperty = dynaClass.getDynaProperty(toString(key));  
123 return (dynaProperty == null ? false : true);  
124 }  
125  
126 /\*\*  
127 \* Indicates whether the decorated {@link DynaBean} contains  
128 \* a specified value.  
129 \*  
130 \* @param value The value to check for.  
131 \* @return <code>true</code> if one of the the {@link DynaBean}'s  
132 \* properties contains the specified value, otherwise  
133 \* <code>false</code>.  
134 \*/  
135 public boolean containsValue(final Object value) {  
136 final DynaProperty[] properties = getDynaProperties();  
137 for (DynaProperty propertie : properties) {  
138 final String key = propertie.getName();  
139 final Object prop = getDynaBean().get(key);  
140 if (value == null) {  
141 if (prop == null) {  
142 return true;  
143 }  
144 } else {  
145 if (value.equals(prop)) {  
146 return true;  
147 }  
148 }  
149 }  
150 return false;  
151 }  
152  
153 /\*\*  
154 \* <p>Returns the Set of the property/value mappings  
155 \* in the decorated {@link DynaBean}.</p>  
156 \*  
157 \* <p>Each element in the Set is a <code>Map.Entry</code>  
158 \* type.</p>  
159 \*  
160 \* @return An unmodifiable set of the DynaBean  
161 \* property name/value pairs  
162 \*/  
163 public Set<Map.Entry<K, Object>> entrySet() {  
164 final DynaProperty[] properties = getDynaProperties();  
165 final Set<Map.Entry<K, Object>> set = new HashSet<Map.Entry<K, Object>>(properties.length);  
166 for (DynaProperty propertie : properties) {  
167 final K key = convertKey(propertie.getName());  
168 final Object value = getDynaBean().get(propertie.getName());  
169 set.add(new MapEntry<K>(key, value));  
170 }  
171 return Collections.unmodifiableSet(set);  
172 }  
173  
174 /\*\*  
175 \* Return the value for the specified key from  
176 \* the decorated {@link DynaBean}.  
177 \*  
178 \* @param key The {@link DynaBean}'s property name  
179 \* @return The value for the specified property.  
180 \*/  
181 public Object get(final Object key) {  
182 return getDynaBean().get(toString(key));  
183 }  
184  
185 /\*\*  
186 \* Indicate whether the decorated {@link DynaBean} has  
187 \* any properties.  
188 \*  
189 \* @return <code>true</code> if the {@link DynaBean} has  
190 \* no properties, otherwise <code>false</code>.  
191 \*/  
192 public boolean isEmpty() {  
193 return (getDynaProperties().length == 0);  
194 }  
195  
196 /\*\*  
197 \* <p>Returns the Set of the property  
198 \* names in the decorated {@link DynaBean}.</p>  
199 \*  
200 \* <p><b>N.B.</b>For {@link DynaBean}s whose associated {@link DynaClass}  
201 \* is a {@link MutableDynaClass} a new Set is created every  
202 \* time, otherwise the Set is created only once and cached.</p>  
203 \*  
204 \* @return An unmodifiable set of the {@link DynaBean}s  
205 \* property names.  
206 \*/  
207 public Set<K> keySet() {  
208 if (keySet != null) {  
209 return keySet;  
210 }  
211  
212 // Create a Set of the keys  
213 final DynaProperty[] properties = getDynaProperties();  
214 Set<K> set = new HashSet<K>(properties.length);  
215 for (DynaProperty propertie : properties) {  
216 set.add(convertKey(propertie.getName()));  
217 }  
218 set = Collections.unmodifiableSet(set);  
219  
220 // Cache the keySet if Not a MutableDynaClass  
221 final DynaClass dynaClass = getDynaBean().getDynaClass();  
222 if (!(dynaClass instanceof MutableDynaClass)) {  
223 keySet = set;  
224 }  
225  
226 return set;  
227  
228 }  
229  
230 /\*\*  
231 \* Set the value for the specified property in  
232 \* the decorated {@link DynaBean}.  
233 \*  
234 \* @param key The {@link DynaBean}'s property name  
235 \* @param value The value for the specified property.  
236 \* @return The previous property's value.  
237 \* @throws UnsupportedOperationException if  
238 \* <code>isReadOnly()</code> is true.  
239 \*/  
240 public Object put(final K key, final Object value) {  
241 if (isReadOnly()) {  
242 throw new UnsupportedOperationException("Map is read only");  
243 }  
244 final String property = toString(key);  
245 final Object previous = getDynaBean().get(property);  
246 getDynaBean().set(property, value);  
247 return previous;  
248 }  
249  
250 /\*\*  
251 \* Copy the contents of a Map to the decorated {@link DynaBean}.  
252 \*  
253 \* @param map The Map of values to copy.  
254 \* @throws UnsupportedOperationException if  
255 \* <code>isReadOnly()</code> is true.  
256 \*/  
257 public void putAll(final Map<? extends K, ? extends Object> map) {  
258 if (isReadOnly()) {  
259 throw new UnsupportedOperationException("Map is read only");  
260 }  
261 for (final Map.Entry<? extends K, ?> e : map.entrySet()) {  
262 put(e.getKey(), e.getValue());  
263 }  
264 }  
265  
266 /\*\*  
267 \* remove() operation is not supported.  
268 \*  
269 \* @param key The {@link DynaBean}'s property name  
270 \* @return the value removed  
271 \* @throws UnsupportedOperationException  
272 \*/  
273 public Object remove(final Object key) {  
274 throw new UnsupportedOperationException();  
275 }  
276  
277 /\*\*  
278 \* Returns the number properties in the decorated  
279 \* {@link DynaBean}.  
280 \* @return The number of properties.  
281 \*/  
282 public int size() {  
283 return getDynaProperties().length;  
284 }  
285  
286 /\*\*  
287 \* Returns the set of property values in the  
288 \* decorated {@link DynaBean}.  
289 \*  
290 \* @return Unmodifiable collection of values.  
291 \*/  
292 public Collection<Object> values() {  
293 final DynaProperty[] properties = getDynaProperties();  
294 final List<Object> values = new ArrayList<Object>(properties.length);  
295 for (DynaProperty propertie : properties) {  
296 final String key = propertie.getName();  
297 final Object value = getDynaBean().get(key);  
298 values.add(value);  
299 }  
300 return Collections.unmodifiableList(values);  
301 }  
302  
303 // ------------------- protected Methods -----------------------------  
304  
305 /\*\*  
306 \* Provide access to the underlying {@link DynaBean}  
307 \* this Map decorates.  
308 \*  
309 \* @return the decorated {@link DynaBean}.  
310 \*/  
311 public DynaBean getDynaBean() {  
312 return dynaBean;  
313 }  
314  
315 /\*\*  
316 \* Converts the name of a property to the key type of this decorator.  
317 \*  
318 \* @param propertyName the name of a property  
319 \* @return the converted key to be used in the decorated map  
320 \*/  
321 protected abstract K convertKey(String propertyName);  
322  
323 // ------------------- private Methods -------------------------------  
324  
325 /\*\*  
326 \* Convenience method to retrieve the {@link DynaProperty}s  
327 \* for this {@link DynaClass}.  
328 \*  
329 \* @return The an array of the {@link DynaProperty}s.  
330 \*/  
331 private DynaProperty[] getDynaProperties() {  
332 return getDynaBean().getDynaClass().getDynaProperties();  
333 }  
334  
335 /\*\*  
336 \* Convenience method to convert an Object  
337 \* to a String.  
338 \*  
339 \* @param obj The Object to convert  
340 \* @return String representation of the object  
341 \*/  
342 private String toString(final Object obj) {  
343 return (obj == null ? null : obj.toString());  
344 }  
345  
346 /\*\*  
347 \* Map.Entry implementation.  
348 \*/  
349 private static class MapEntry<K> implements Map.Entry<K, Object> {  
350 private final K key;  
351 private final Object value;  
352 MapEntry(final K key, final Object value) {  
353 this.key = key;  
354 this.value = value;  
355 }  
356 @Override  
357 public boolean equals(final Object o) {  
358 if (!(o instanceof Map.Entry)) {  
359 return false;  
360 }  
361 final Map.Entry<?, ?> e = (Map.Entry<?, ?>)o;  
362 return ((key.equals(e.getKey())) &&  
363 (value == null ? e.getValue() == null  
364 : value.equals(e.getValue())));  
365 }  
366 @Override  
367 public int hashCode() {  
368 return key.hashCode() + (value == null ? 0 : value.hashCode());  
369 }  
370 public K getKey() {  
371 return key;  
372 }  
373 public Object getValue() {  
374 return value;  
375 }  
376 public Object setValue(final Object value) {  
377 throw new UnsupportedOperationException();  
378 }  
379 }  
380  
381}