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.IOException;  
023import java.io.ObjectInputStream;  
024import java.io.ObjectOutputStream;  
025import java.io.Serializable;  
026import java.io.StreamCorruptedException;  
027import java.util.List;  
028import java.util.Map;  
029  
030  
031/\*\*  
032 \* <p>The metadata describing an individual property of a DynaBean.</p>  
033 \*  
034 \* <p>The meta contains an <em>optional</em> content type property ({@link #getContentType})  
035 \* for use by mapped and iterated properties.  
036 \* A mapped or iterated property may choose to indicate the type it expects.  
037 \* The DynaBean implementation may choose to enforce this type on its entries.  
038 \* Alternatively, an implementatin may choose to ignore this property.  
039 \* All keys for maps must be of type String so no meta data is needed for map keys.</p>  
040 \*  
041 \* @version $Id$  
042 \*/  
043  
044public class DynaProperty implements Serializable {  
045  
046 // ----------------------------------------------------------- Constants  
047  
048 /\*  
049 \* There are issues with serializing primitive class types on certain JVM versions  
050 \* (including java 1.3).  
051 \* This class uses a custom serialization implementation that writes an integer  
052 \* for these primitive class.  
053 \* This list of constants are the ones used in serialization.  
054 \* If these values are changed, then older versions will no longer be read correctly  
055 \*/  
056 private static final int BOOLEAN\_TYPE = 1;  
057 private static final int BYTE\_TYPE = 2;  
058 private static final int CHAR\_TYPE = 3;  
059 private static final int DOUBLE\_TYPE = 4;  
060 private static final int FLOAT\_TYPE = 5;  
061 private static final int INT\_TYPE = 6;  
062 private static final int LONG\_TYPE = 7;  
063 private static final int SHORT\_TYPE = 8;  
064  
065  
066 // ----------------------------------------------------------- Constructors  
067  
068  
069 /\*\*  
070 \* Construct a property that accepts any data type.  
071 \*  
072 \* @param name Name of the property being described  
073 \*/  
074 public DynaProperty(final String name) {  
075  
076 this(name, Object.class);  
077  
078 }  
079  
080  
081 /\*\*  
082 \* Construct a property of the specified data type.  
083 \*  
084 \* @param name Name of the property being described  
085 \* @param type Java class representing the property data type  
086 \*/  
087 public DynaProperty(final String name, final Class<?> type) {  
088  
089 super();  
090 this.name = name;  
091 this.type = type;  
092 if (type != null && type.isArray()) {  
093 this.contentType = type.getComponentType();  
094 }  
095  
096 }  
097  
098 /\*\*  
099 \* Construct an indexed or mapped <code>DynaProperty</code> that supports (pseudo)-introspection  
100 \* of the content type.  
101 \*  
102 \* @param name Name of the property being described  
103 \* @param type Java class representing the property data type  
104 \* @param contentType Class that all indexed or mapped elements are instances of  
105 \*/  
106 public DynaProperty(final String name, final Class<?> type, final Class<?> contentType) {  
107  
108 super();  
109 this.name = name;  
110 this.type = type;  
111 this.contentType = contentType;  
112  
113 }  
114  
115 // ------------------------------------------------------------- Properties  
116  
117 /\*\* Property name \*/  
118 protected String name = null;  
119 /\*\*  
120 \* Get the name of this property.  
121 \* @return the name of the property  
122 \*/  
123 public String getName() {  
124 return (this.name);  
125 }  
126  
127 /\*\* Property type \*/  
128 protected transient Class<?> type = null;  
129 /\*\*  
130 \* <p>Gets the Java class representing the data type of the underlying property  
131 \* values.</p>  
132 \*  
133 \* <p>There are issues with serializing primitive class types on certain JVM versions  
134 \* (including java 1.3).  
135 \* Therefore, this field <strong>must not be serialized using the standard methods</strong>.</p>  
136 \*  
137 \* <p><strong>Please leave this field as <code>transient</code></strong></p>  
138 \*  
139 \* @return the property type  
140 \*/  
141 public Class<?> getType() {  
142 return (this.type);  
143 }  
144  
145  
146 /\*\* The <em>(optional)</em> type of content elements for indexed <code>DynaProperty</code> \*/  
147 protected transient Class<?> contentType;  
148 /\*\*  
149 \* Gets the <em>(optional)</em> type of the indexed content for <code>DynaProperty</code>'s  
150 \* that support this feature.  
151 \*  
152 \* <p>There are issues with serializing primitive class types on certain JVM versions  
153 \* (including java 1.3).  
154 \* Therefore, this field <strong>must not be serialized using the standard methods</strong>.</p>  
155 \*  
156 \* @return the Class for the content type if this is an indexed <code>DynaProperty</code>  
157 \* and this feature is supported. Otherwise null.  
158 \*/  
159 public Class<?> getContentType() {  
160 return contentType;  
161 }  
162  
163 // --------------------------------------------------------- Public Methods  
164  
165  
166 /\*\*  
167 \* Does this property represent an indexed value (ie an array or List)?  
168 \*  
169 \* @return <code>true</code> if the property is indexed (i.e. is a List or  
170 \* array), otherwise <code>false</code>  
171 \*/  
172 public boolean isIndexed() {  
173  
174 if (type == null) {  
175 return (false);  
176 } else if (type.isArray()) {  
177 return (true);  
178 } else if (List.class.isAssignableFrom(type)) {  
179 return (true);  
180 } else {  
181 return (false);  
182 }  
183  
184 }  
185  
186  
187 /\*\*  
188 \* Does this property represent a mapped value (ie a Map)?  
189 \*  
190 \* @return <code>true</code> if the property is a Map  
191 \* otherwise <code>false</code>  
192 \*/  
193 public boolean isMapped() {  
194  
195 if (type == null) {  
196 return (false);  
197 } else {  
198 return (Map.class.isAssignableFrom(type));  
199 }  
200  
201 }  
202  
203 /\*\*  
204 \* Checks this instance against the specified Object for equality. Overrides the  
205 \* default refererence test for equality provided by {@link java.lang.Object#equals(Object)}  
206 \* @param obj The object to compare to  
207 \* @return <code>true</code> if object is a dyna property with the same name  
208 \* type and content type, otherwise <code>false</code>  
209 \* @since 1.8.0  
210 \*/  
211 @Override  
212 public boolean equals(final Object obj) {  
213  
214 boolean result = false;  
215  
216 result = (obj == this);  
217  
218 if ((!result) && obj instanceof DynaProperty) {  
219 final DynaProperty that = (DynaProperty) obj;  
220 result =  
221 ((this.name == null) ? (that.name == null) : (this.name.equals(that.name))) &&  
222 ((this.type == null) ? (that.type == null) : (this.type.equals(that.type))) &&  
223 ((this.contentType == null) ? (that.contentType == null) : (this.contentType.equals(that.contentType)));  
224 }  
225  
226 return result;  
227 }  
228  
229 /\*\*  
230 \* @return the hashcode for this dyna property  
231 \* @see java.lang.Object#hashCode  
232 \* @since 1.8.0  
233 \*/  
234 @Override  
235 public int hashCode() {  
236  
237 int result = 1;  
238  
239 result = result \* 31 + ((name == null) ? 0 : name.hashCode());  
240 result = result \* 31 + ((type == null) ? 0 : type.hashCode());  
241 result = result \* 31 + ((contentType == null) ? 0 : contentType.hashCode());  
242  
243 return result;  
244 }  
245  
246 /\*\*  
247 \* Return a String representation of this Object.  
248 \* @return a String representation of the dyna property  
249 \*/  
250 @Override  
251 public String toString() {  
252  
253 final StringBuilder sb = new StringBuilder("DynaProperty[name=");  
254 sb.append(this.name);  
255 sb.append(",type=");  
256 sb.append(this.type);  
257 if (isMapped() || isIndexed()) {  
258 sb.append(" <").append(this.contentType).append(">");  
259 }  
260 sb.append("]");  
261 return (sb.toString());  
262  
263 }  
264  
265 // --------------------------------------------------------- Serialization helper methods  
266  
267 /\*\*  
268 \* Writes this object safely.  
269 \* There are issues with serializing primitive class types on certain JVM versions  
270 \* (including java 1.3).  
271 \* This method provides a workaround.  
272 \*/  
273 private void writeObject(final ObjectOutputStream out) throws IOException {  
274  
275 writeAnyClass(this.type,out);  
276  
277 if (isMapped() || isIndexed()) {  
278 writeAnyClass(this.contentType,out);  
279 }  
280  
281 // write out other values  
282 out.defaultWriteObject();  
283 }  
284  
285 /\*\*  
286 \* Write a class using safe encoding to workaround java 1.3 serialization bug.  
287 \*/  
288 private void writeAnyClass(final Class<?> clazz, final ObjectOutputStream out) throws IOException {  
289 // safely write out any class  
290 int primitiveType = 0;  
291 if (Boolean.TYPE.equals(clazz)) {  
292 primitiveType = BOOLEAN\_TYPE;  
293 } else if (Byte.TYPE.equals(clazz)) {  
294 primitiveType = BYTE\_TYPE;  
295 } else if (Character.TYPE.equals(clazz)) {  
296 primitiveType = CHAR\_TYPE;  
297 } else if (Double.TYPE.equals(clazz)) {  
298 primitiveType = DOUBLE\_TYPE;  
299 } else if (Float.TYPE.equals(clazz)) {  
300 primitiveType = FLOAT\_TYPE;  
301 } else if (Integer.TYPE.equals(clazz)) {  
302 primitiveType = INT\_TYPE;  
303 } else if (Long.TYPE.equals(clazz)) {  
304 primitiveType = LONG\_TYPE;  
305 } else if (Short.TYPE.equals(clazz)) {  
306 primitiveType = SHORT\_TYPE;  
307 }  
308  
309 if (primitiveType == 0) {  
310 // then it's not a primitive type  
311 out.writeBoolean(false);  
312 out.writeObject(clazz);  
313 } else {  
314 // we'll write out a constant instead  
315 out.writeBoolean(true);  
316 out.writeInt(primitiveType);  
317 }  
318 }  
319  
320 /\*\*  
321 \* Reads field values for this object safely.  
322 \* There are issues with serializing primitive class types on certain JVM versions  
323 \* (including java 1.3).  
324 \* This method provides a workaround.  
325 \*  
326 \* @throws StreamCorruptedException when the stream data values are outside expected range  
327 \*/  
328 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
329  
330 this.type = readAnyClass(in);  
331  
332 if (isMapped() || isIndexed()) {  
333 this.contentType = readAnyClass(in);  
334 }  
335  
336 // read other values  
337 in.defaultReadObject();  
338 }  
339  
340  
341 /\*\*  
342 \* Reads a class using safe encoding to workaround java 1.3 serialization bug.  
343 \*/  
344 private Class<?> readAnyClass(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
345 // read back type class safely  
346 if (in.readBoolean()) {  
347 // it's a type constant  
348 switch (in.readInt()) {  
349  
350 case BOOLEAN\_TYPE: return Boolean.TYPE;  
351 case BYTE\_TYPE: return Byte.TYPE;  
352 case CHAR\_TYPE: return Character.TYPE;  
353 case DOUBLE\_TYPE: return Double.TYPE;  
354 case FLOAT\_TYPE: return Float.TYPE;  
355 case INT\_TYPE: return Integer.TYPE;  
356 case LONG\_TYPE: return Long.TYPE;  
357 case SHORT\_TYPE: return Short.TYPE;  
358 default:  
359 // something's gone wrong  
360 throw new StreamCorruptedException(  
361 "Invalid primitive type. "  
362 + "Check version of beanutils used to serialize is compatible.");  
363  
364 }  
365  
366 } else {  
367 // it's another class  
368 return ((Class<?>) in.readObject());  
369 }  
370 }  
371}