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  
018package org.apache.commons.beanutils;  
019  
020  
021import java.beans.IntrospectionException;  
022import java.beans.PropertyDescriptor;  
023import java.lang.ref.Reference;  
024import java.lang.ref.SoftReference;  
025import java.lang.ref.WeakReference;  
026import java.lang.reflect.Method;  
027import java.lang.reflect.Modifier;  
028  
029  
030/\*\*  
031 \* A MappedPropertyDescriptor describes one mapped property.  
032 \* Mapped properties are multivalued properties like indexed properties  
033 \* but that are accessed with a String key instead of an index.  
034 \* Such property values are typically stored in a Map collection.  
035 \* For this class to work properly, a mapped value must have  
036 \* getter and setter methods of the form  
037 \* <p><code>get<strong>Property</strong>(String key)</code> and  
038 \* <p><code>set<strong>Property</strong>(String key, Object value)</code>,  
039 \* <p>where <code><strong>Property</strong></code> must be replaced  
040 \* by the name of the property.  
041 \* @see java.beans.PropertyDescriptor  
042 \*  
043 \* @version $Id$  
044 \*/  
045public class MappedPropertyDescriptor extends PropertyDescriptor {  
046 // ----------------------------------------------------- Instance Variables  
047  
048 /\*\*  
049 \* The underlying data type of the property we are describing.  
050 \*/  
051 private Reference<Class<?>> mappedPropertyTypeRef;  
052  
053 /\*\*  
054 \* The reader method for this property (if any).  
055 \*/  
056 private MappedMethodReference mappedReadMethodRef;  
057  
058 /\*\*  
059 \* The writer method for this property (if any).  
060 \*/  
061 private MappedMethodReference mappedWriteMethodRef;  
062  
063 /\*\*  
064 \* The parameter types array for the reader method signature.  
065 \*/  
066 private static final Class<?>[] STRING\_CLASS\_PARAMETER = new Class[]{String.class};  
067  
068 // ----------------------------------------------------------- Constructors  
069  
070 /\*\*  
071 \* Constructs a MappedPropertyDescriptor for a property that follows  
072 \* the standard Java convention by having getFoo and setFoo  
073 \* accessor methods, with the addition of a String parameter (the key).  
074 \* Thus if the argument name is "fred", it will  
075 \* assume that the writer method is "setFred" and the reader method  
076 \* is "getFred". Note that the property name should start with a lower  
077 \* case character, which will be capitalized in the method names.  
078 \*  
079 \* @param propertyName The programmatic name of the property.  
080 \* @param beanClass The Class object for the target bean. For  
081 \* example sun.beans.OurButton.class.  
082 \*  
083 \* @throws IntrospectionException if an exception occurs during  
084 \* introspection.  
085 \*/  
086 public MappedPropertyDescriptor(final String propertyName, final Class<?> beanClass)  
087 throws IntrospectionException {  
088  
089 super(propertyName, null, null);  
090  
091 if (propertyName == null || propertyName.length() == 0) {  
092 throw new IntrospectionException("bad property name: " +  
093 propertyName + " on class: " + beanClass.getClass().getName());  
094 }  
095  
096 setName(propertyName);  
097 final String base = capitalizePropertyName(propertyName);  
098  
099 // Look for mapped read method and matching write method  
100 Method mappedReadMethod = null;  
101 Method mappedWriteMethod = null;  
102 try {  
103 try {  
104 mappedReadMethod = getMethod(beanClass, "get" + base,  
105 STRING\_CLASS\_PARAMETER);  
106 } catch (final IntrospectionException e) {  
107 mappedReadMethod = getMethod(beanClass, "is" + base,  
108 STRING\_CLASS\_PARAMETER);  
109 }  
110 final Class<?>[] params = { String.class, mappedReadMethod.getReturnType() };  
111 mappedWriteMethod = getMethod(beanClass, "set" + base, params);  
112 } catch (final IntrospectionException e) {  
113 /\* Swallow IntrospectionException  
114 \* TODO: Why?  
115 \*/  
116 }  
117  
118 // If there's no read method, then look for just a write method  
119 if (mappedReadMethod == null) {  
120 mappedWriteMethod = getMethod(beanClass, "set" + base, 2);  
121 }  
122  
123 if ((mappedReadMethod == null) && (mappedWriteMethod == null)) {  
124 throw new IntrospectionException("Property '" + propertyName +  
125 "' not found on " +  
126 beanClass.getName());  
127 }  
128 mappedReadMethodRef = new MappedMethodReference(mappedReadMethod);  
129 mappedWriteMethodRef = new MappedMethodReference(mappedWriteMethod);  
130  
131 findMappedPropertyType();  
132 }  
133  
134  
135 /\*\*  
136 \* This constructor takes the name of a mapped property, and method  
137 \* names for reading and writing the property.  
138 \*  
139 \* @param propertyName The programmatic name of the property.  
140 \* @param beanClass The Class object for the target bean. For  
141 \* example sun.beans.OurButton.class.  
142 \* @param mappedGetterName The name of the method used for  
143 \* reading one of the property values. May be null if the  
144 \* property is write-only.  
145 \* @param mappedSetterName The name of the method used for writing  
146 \* one of the property values. May be null if the property is  
147 \* read-only.  
148 \*  
149 \* @throws IntrospectionException if an exception occurs during  
150 \* introspection.  
151 \*/  
152 public MappedPropertyDescriptor(final String propertyName, final Class<?> beanClass,  
153 final String mappedGetterName, final String mappedSetterName)  
154 throws IntrospectionException {  
155  
156 super(propertyName, null, null);  
157  
158 if (propertyName == null || propertyName.length() == 0) {  
159 throw new IntrospectionException("bad property name: " +  
160 propertyName);  
161 }  
162 setName(propertyName);  
163  
164 // search the mapped get and set methods  
165 Method mappedReadMethod = null;  
166 Method mappedWriteMethod = null;  
167 mappedReadMethod =  
168 getMethod(beanClass, mappedGetterName, STRING\_CLASS\_PARAMETER);  
169  
170 if (mappedReadMethod != null) {  
171 final Class<?>[] params = { String.class, mappedReadMethod.getReturnType() };  
172 mappedWriteMethod =  
173 getMethod(beanClass, mappedSetterName, params);  
174 } else {  
175 mappedWriteMethod =  
176 getMethod(beanClass, mappedSetterName, 2);  
177 }  
178 mappedReadMethodRef = new MappedMethodReference(mappedReadMethod);  
179 mappedWriteMethodRef = new MappedMethodReference(mappedWriteMethod);  
180  
181 findMappedPropertyType();  
182 }  
183  
184 /\*\*  
185 \* This constructor takes the name of a mapped property, and Method  
186 \* objects for reading and writing the property.  
187 \*  
188 \* @param propertyName The programmatic name of the property.  
189 \* @param mappedGetter The method used for reading one of  
190 \* the property values. May be be null if the property  
191 \* is write-only.  
192 \* @param mappedSetter The method used for writing one the  
193 \* property values. May be null if the property is read-only.  
194 \*  
195 \* @throws IntrospectionException if an exception occurs during  
196 \* introspection.  
197 \*/  
198 public MappedPropertyDescriptor(final String propertyName,  
199 final Method mappedGetter, final Method mappedSetter)  
200 throws IntrospectionException {  
201  
202 super(propertyName, mappedGetter, mappedSetter);  
203  
204 if (propertyName == null || propertyName.length() == 0) {  
205 throw new IntrospectionException("bad property name: " +  
206 propertyName);  
207 }  
208  
209 setName(propertyName);  
210 mappedReadMethodRef = new MappedMethodReference(mappedGetter);  
211 mappedWriteMethodRef = new MappedMethodReference(mappedSetter);  
212 findMappedPropertyType();  
213 }  
214  
215 // -------------------------------------------------------- Public Methods  
216  
217 /\*\*  
218 \* Gets the Class object for the property values.  
219 \*  
220 \* @return The Java type info for the property values. Note that  
221 \* the "Class" object may describe a built-in Java type such as "int".  
222 \* The result may be "null" if this is a mapped property that  
223 \* does not support non-keyed access.  
224 \* <p>  
225 \* This is the type that will be returned by the mappedReadMethod.  
226 \*/  
227 public Class<?> getMappedPropertyType() {  
228 return mappedPropertyTypeRef.get();  
229 }  
230  
231 /\*\*  
232 \* Gets the method that should be used to read one of the property value.  
233 \*  
234 \* @return The method that should be used to read the property value.  
235 \* May return null if the property can't be read.  
236 \*/  
237 public Method getMappedReadMethod() {  
238 return mappedReadMethodRef.get();  
239 }  
240  
241 /\*\*  
242 \* Sets the method that should be used to read one of the property value.  
243 \*  
244 \* @param mappedGetter The mapped getter method.  
245 \* @throws IntrospectionException If an error occurs finding the  
246 \* mapped property  
247 \*/  
248 public void setMappedReadMethod(final Method mappedGetter)  
249 throws IntrospectionException {  
250 mappedReadMethodRef = new MappedMethodReference(mappedGetter);  
251 findMappedPropertyType();  
252 }  
253  
254 /\*\*  
255 \* Gets the method that should be used to write one of the property value.  
256 \*  
257 \* @return The method that should be used to write one of the property value.  
258 \* May return null if the property can't be written.  
259 \*/  
260 public Method getMappedWriteMethod() {  
261 return mappedWriteMethodRef.get();  
262 }  
263  
264 /\*\*  
265 \* Sets the method that should be used to write the property value.  
266 \*  
267 \* @param mappedSetter The mapped setter method.  
268 \* @throws IntrospectionException If an error occurs finding the  
269 \* mapped property  
270 \*/  
271 public void setMappedWriteMethod(final Method mappedSetter)  
272 throws IntrospectionException {  
273 mappedWriteMethodRef = new MappedMethodReference(mappedSetter);  
274 findMappedPropertyType();  
275 }  
276  
277 // ------------------------------------------------------- Private Methods  
278  
279 /\*\*  
280 \* Introspect our bean class to identify the corresponding getter  
281 \* and setter methods.  
282 \*/  
283 private void findMappedPropertyType() throws IntrospectionException {  
284 try {  
285 final Method mappedReadMethod = getMappedReadMethod();  
286 final Method mappedWriteMethod = getMappedWriteMethod();  
287 Class<?> mappedPropertyType = null;  
288 if (mappedReadMethod != null) {  
289 if (mappedReadMethod.getParameterTypes().length != 1) {  
290 throw new IntrospectionException  
291 ("bad mapped read method arg count");  
292 }  
293 mappedPropertyType = mappedReadMethod.getReturnType();  
294 if (mappedPropertyType == Void.TYPE) {  
295 throw new IntrospectionException  
296 ("mapped read method " +  
297 mappedReadMethod.getName() + " returns void");  
298 }  
299 }  
300  
301 if (mappedWriteMethod != null) {  
302 final Class<?>[] params = mappedWriteMethod.getParameterTypes();  
303 if (params.length != 2) {  
304 throw new IntrospectionException  
305 ("bad mapped write method arg count");  
306 }  
307 if (mappedPropertyType != null &&  
308 mappedPropertyType != params[1]) {  
309 throw new IntrospectionException  
310 ("type mismatch between mapped read and write methods");  
311 }  
312 mappedPropertyType = params[1];  
313 }  
314 mappedPropertyTypeRef = new SoftReference<Class<?>>(mappedPropertyType);  
315 } catch (final IntrospectionException ex) {  
316 throw ex;  
317 }  
318 }  
319  
320  
321 /\*\*  
322 \* Return a capitalized version of the specified property name.  
323 \*  
324 \* @param s The property name  
325 \*/  
326 private static String capitalizePropertyName(final String s) {  
327 if (s.length() == 0) {  
328 return s;  
329 }  
330  
331 final char[] chars = s.toCharArray();  
332 chars[0] = Character.toUpperCase(chars[0]);  
333 return new String(chars);  
334 }  
335  
336 /\*\*  
337 \* Find a method on a class with a specified number of parameters.  
338 \*/  
339 private static Method internalGetMethod(final Class<?> initial, final String methodName,  
340 final int parameterCount) {  
341 // For overridden methods we need to find the most derived version.  
342 // So we start with the given class and walk up the superclass chain.  
343 for (Class<?> clazz = initial; clazz != null; clazz = clazz.getSuperclass()) {  
344 final Method[] methods = clazz.getDeclaredMethods();  
345 for (final Method method : methods) {  
346 if (method == null) {  
347 continue;  
348 }  
349 // skip static methods.  
350 final int mods = method.getModifiers();  
351 if (!Modifier.isPublic(mods) ||  
352 Modifier.isStatic(mods)) {  
353 continue;  
354 }  
355 if (method.getName().equals(methodName) &&  
356 method.getParameterTypes().length == parameterCount) {  
357 return method;  
358 }  
359 }  
360 }  
361  
362 // Now check any inherited interfaces. This is necessary both when  
363 // the argument class is itself an interface, and when the argument  
364 // class is an abstract class.  
365 final Class<?>[] interfaces = initial.getInterfaces();  
366 for (Class<?> interface1 : interfaces) {  
367 final Method method = internalGetMethod(interface1, methodName, parameterCount);  
368 if (method != null) {  
369 return method;  
370 }  
371 }  
372  
373 return null;  
374 }  
375  
376 /\*\*  
377 \* Find a method on a class with a specified number of parameters.  
378 \*/  
379 private static Method getMethod(final Class<?> clazz, final String methodName, final int parameterCount)  
380 throws IntrospectionException {  
381 if (methodName == null) {  
382 return null;  
383 }  
384  
385 final Method method = internalGetMethod(clazz, methodName, parameterCount);  
386 if (method != null) {  
387 return method;  
388 }  
389  
390 // No Method found  
391 throw new IntrospectionException("No method \"" + methodName +  
392 "\" with " + parameterCount + " parameter(s)");  
393 }  
394  
395 /\*\*  
396 \* Find a method on a class with a specified parameter list.  
397 \*/  
398 private static Method getMethod(final Class<?> clazz, final String methodName, final Class<?>[] parameterTypes)  
399 throws IntrospectionException {  
400 if (methodName == null) {  
401 return null;  
402 }  
403  
404 final Method method = MethodUtils.getMatchingAccessibleMethod(clazz, methodName, parameterTypes);  
405 if (method != null) {  
406 return method;  
407 }  
408  
409 final int parameterCount = (parameterTypes == null) ? 0 : parameterTypes.length;  
410  
411 // No Method found  
412 throw new IntrospectionException("No method \"" + methodName +  
413 "\" with " + parameterCount + " parameter(s) of matching types.");  
414 }  
415  
416 /\*\*  
417 \* Holds a {@link Method} in a {@link SoftReference} so that it  
418 \* it doesn't prevent any ClassLoader being garbage collected, but  
419 \* tries to re-create the method if the method reference has been  
420 \* released.  
421 \*  
422 \* See http://issues.apache.org/jira/browse/BEANUTILS-291  
423 \*/  
424 private static class MappedMethodReference {  
425 private String className;  
426 private String methodName;  
427 private Reference<Method> methodRef;  
428 private Reference<Class<?>> classRef;  
429 private Reference<Class<?>> writeParamTypeRef0;  
430 private Reference<Class<?>> writeParamTypeRef1;  
431 private String[] writeParamClassNames;  
432 MappedMethodReference(final Method m) {  
433 if (m != null) {  
434 className = m.getDeclaringClass().getName();  
435 methodName = m.getName();  
436 methodRef = new SoftReference<Method>(m);  
437 classRef = new WeakReference<Class<?>>(m.getDeclaringClass());  
438 final Class<?>[] types = m.getParameterTypes();  
439 if (types.length == 2) {  
440 writeParamTypeRef0 = new WeakReference<Class<?>>(types[0]);  
441 writeParamTypeRef1 = new WeakReference<Class<?>>(types[1]);  
442 writeParamClassNames = new String[2];  
443 writeParamClassNames[0] = types[0].getName();  
444 writeParamClassNames[1] = types[1].getName();  
445 }  
446 }  
447 }  
448 private Method get() {  
449 if (methodRef == null) {  
450 return null;  
451 }  
452 Method m = methodRef.get();  
453 if (m == null) {  
454 Class<?> clazz = classRef.get();  
455 if (clazz == null) {  
456 clazz = reLoadClass();  
457 if (clazz != null) {  
458 classRef = new WeakReference<Class<?>>(clazz);  
459 }  
460 }  
461 if (clazz == null) {  
462 throw new RuntimeException("Method " + methodName + " for " +  
463 className + " could not be reconstructed - class reference has gone");  
464 }  
465 Class<?>[] paramTypes = null;  
466 if (writeParamClassNames != null) {  
467 paramTypes = new Class[2];  
468 paramTypes[0] = writeParamTypeRef0.get();  
469 if (paramTypes[0] == null) {  
470 paramTypes[0] = reLoadClass(writeParamClassNames[0]);  
471 if (paramTypes[0] != null) {  
472 writeParamTypeRef0 = new WeakReference<Class<?>>(paramTypes[0]);  
473 }  
474 }  
475 paramTypes[1] = writeParamTypeRef1.get();  
476 if (paramTypes[1] == null) {  
477 paramTypes[1] = reLoadClass(writeParamClassNames[1]);  
478 if (paramTypes[1] != null) {  
479 writeParamTypeRef1 = new WeakReference<Class<?>>(paramTypes[1]);  
480 }  
481 }  
482 } else {  
483 paramTypes = STRING\_CLASS\_PARAMETER;  
484 }  
485 try {  
486 m = clazz.getMethod(methodName, paramTypes);  
487 // Un-comment following line for testing  
488 // System.out.println("Recreated Method " + methodName + " for " + className);  
489 } catch (final NoSuchMethodException e) {  
490 throw new RuntimeException("Method " + methodName + " for " +  
491 className + " could not be reconstructed - method not found");  
492 }  
493 methodRef = new SoftReference<Method>(m);  
494 }  
495 return m;  
496 }  
497  
498 /\*\*  
499 \* Try to re-load the class  
500 \*/  
501 private Class<?> reLoadClass() {  
502 return reLoadClass(className);  
503 }  
504  
505 /\*\*  
506 \* Try to re-load the class  
507 \*/  
508 private Class<?> reLoadClass(final String name) {  
509  
510 ClassLoader classLoader = Thread.currentThread().getContextClassLoader();  
511  
512 // Try the context class loader  
513 if (classLoader != null) {  
514 try {  
515 return classLoader.loadClass(name);  
516 } catch (final ClassNotFoundException e) {  
517 // ignore  
518 }  
519 }  
520  
521 // Try this class's class loader  
522 classLoader = MappedPropertyDescriptor.class.getClassLoader();  
523 try {  
524 return classLoader.loadClass(name);  
525 } catch (final ClassNotFoundException e) {  
526 return null;  
527 }  
528 }  
529 }  
530}