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.expression;  
018  
019/\*\*  
020 \* Default Property Name Expression {@link Resolver} Implementation.  
021 \* <p>  
022 \* This class assists in resolving property names in the following five formats,  
023 \* with the layout of an identifying String in parentheses:  
024 \* <ul>  
025 \* <li><strong>Simple (<code>name</code>)</strong> - The specified  
026 \* <code>name</code> identifies an individual property of a particular  
027 \* JavaBean. The name of the actual getter or setter method to be used  
028 \* is determined using standard JavaBeans instrospection, so that (unless  
029 \* overridden by a <code>BeanInfo</code> class, a property named "xyz"  
030 \* will have a getter method named <code>getXyz()</code> or (for boolean  
031 \* properties only) <code>isXyz()</code>, and a setter method named  
032 \* <code>setXyz()</code>.</li>  
033 \* <li><strong>Nested (<code>name1.name2.name3</code>)</strong> The first  
034 \* name element is used to select a property getter, as for simple  
035 \* references above. The object returned for this property is then  
036 \* consulted, using the same approach, for a property getter for a  
037 \* property named <code>name2</code>, and so on. The property value that  
038 \* is ultimately retrieved or modified is the one identified by the  
039 \* last name element.</li>  
040 \* <li><strong>Indexed (<code>name[index]</code>)</strong> - The underlying  
041 \* property value is assumed to be an array, or this JavaBean is assumed  
042 \* to have indexed property getter and setter methods. The appropriate  
043 \* (zero-relative) entry in the array is selected. <code>List</code>  
044 \* objects are now also supported for read/write. You simply need to define  
045 \* a getter that returns the <code>List</code></li>  
046 \* <li><strong>Mapped (<code>name(key)</code>)</strong> - The JavaBean  
047 \* is assumed to have an property getter and setter methods with an  
048 \* additional attribute of type <code>java.lang.String</code>.</li>  
049 \* <li><strong>Combined (<code>name1.name2[index].name3(key)</code>)</strong> -  
050 \* Combining mapped, nested, and indexed references is also  
051 \* supported.</li>  
052 \* </ul>  
053 \*  
054 \* @version $Id$  
055 \* @since 1.8.0  
056 \*/  
057public class DefaultResolver implements Resolver {  
058  
059 private static final char NESTED = '.';  
060 private static final char MAPPED\_START = '(';  
061 private static final char MAPPED\_END = ')';  
062 private static final char INDEXED\_START = '[';  
063 private static final char INDEXED\_END = ']';  
064  
065 /\*\*  
066 \* Default Constructor.  
067 \*/  
068 public DefaultResolver() {  
069 }  
070  
071 /\*\*  
072 \* Return the index value from the property expression or -1.  
073 \*  
074 \* @param expression The property expression  
075 \* @return The index value or -1 if the property is not indexed  
076 \* @throws IllegalArgumentException If the indexed property is illegally  
077 \* formed or has an invalid (non-numeric) value.  
078 \*/  
079 public int getIndex(final String expression) {  
080 if (expression == null || expression.length() == 0) {  
081 return -1;  
082 }  
083 for (int i = 0; i < expression.length(); i++) {  
084 final char c = expression.charAt(i);  
085 if (c == NESTED || c == MAPPED\_START) {  
086 return -1;  
087 } else if (c == INDEXED\_START) {  
088 final int end = expression.indexOf(INDEXED\_END, i);  
089 if (end < 0) {  
090 throw new IllegalArgumentException("Missing End Delimiter");  
091 }  
092 final String value = expression.substring(i + 1, end);  
093 if (value.length() == 0) {  
094 throw new IllegalArgumentException("No Index Value");  
095 }  
096 int index = 0;  
097 try {  
098 index = Integer.parseInt(value, 10);  
099 } catch (final Exception e) {  
100 throw new IllegalArgumentException("Invalid index value '"  
101 + value + "'");  
102 }  
103 return index;  
104 }  
105 }  
106 return -1;  
107 }  
108  
109 /\*\*  
110 \* Return the map key from the property expression or <code>null</code>.  
111 \*  
112 \* @param expression The property expression  
113 \* @return The index value  
114 \* @throws IllegalArgumentException If the mapped property is illegally formed.  
115 \*/  
116 public String getKey(final String expression) {  
117 if (expression == null || expression.length() == 0) {  
118 return null;  
119 }  
120 for (int i = 0; i < expression.length(); i++) {  
121 final char c = expression.charAt(i);  
122 if (c == NESTED || c == INDEXED\_START) {  
123 return null;  
124 } else if (c == MAPPED\_START) {  
125 final int end = expression.indexOf(MAPPED\_END, i);  
126 if (end < 0) {  
127 throw new IllegalArgumentException("Missing End Delimiter");  
128 }  
129 return expression.substring(i + 1, end);  
130 }  
131 }  
132 return null;  
133 }  
134  
135 /\*\*  
136 \* Return the property name from the property expression.  
137 \*  
138 \* @param expression The property expression  
139 \* @return The property name  
140 \*/  
141 public String getProperty(final String expression) {  
142 if (expression == null || expression.length() == 0) {  
143 return expression;  
144 }  
145 for (int i = 0; i < expression.length(); i++) {  
146 final char c = expression.charAt(i);  
147 if (c == NESTED) {  
148 return expression.substring(0, i);  
149 } else if (c == MAPPED\_START || c == INDEXED\_START) {  
150 return expression.substring(0, i);  
151 }  
152 }  
153 return expression;  
154 }  
155  
156 /\*\*  
157 \* Indicates whether or not the expression  
158 \* contains nested property expressions or not.  
159 \*  
160 \* @param expression The property expression  
161 \* @return The next property expression  
162 \*/  
163 public boolean hasNested(final String expression) {  
164 if (expression == null || expression.length() == 0) {  
165 return false;  
166 } else {  
167 return (remove(expression) != null);  
168 }  
169 }  
170  
171 /\*\*  
172 \* Indicate whether the expression is for an indexed property or not.  
173 \*  
174 \* @param expression The property expression  
175 \* @return <code>true</code> if the expresion is indexed,  
176 \* otherwise <code>false</code>  
177 \*/  
178 public boolean isIndexed(final String expression) {  
179 if (expression == null || expression.length() == 0) {  
180 return false;  
181 }  
182 for (int i = 0; i < expression.length(); i++) {  
183 final char c = expression.charAt(i);  
184 if (c == NESTED || c == MAPPED\_START) {  
185 return false;  
186 } else if (c == INDEXED\_START) {  
187 return true;  
188 }  
189 }  
190 return false;  
191 }  
192  
193 /\*\*  
194 \* Indicate whether the expression is for a mapped property or not.  
195 \*  
196 \* @param expression The property expression  
197 \* @return <code>true</code> if the expresion is mapped,  
198 \* otherwise <code>false</code>  
199 \*/  
200 public boolean isMapped(final String expression) {  
201 if (expression == null || expression.length() == 0) {  
202 return false;  
203 }  
204 for (int i = 0; i < expression.length(); i++) {  
205 final char c = expression.charAt(i);  
206 if (c == NESTED || c == INDEXED\_START) {  
207 return false;  
208 } else if (c == MAPPED\_START) {  
209 return true;  
210 }  
211 }  
212 return false;  
213 }  
214  
215 /\*\*  
216 \* Extract the next property expression from the  
217 \* current expression.  
218 \*  
219 \* @param expression The property expression  
220 \* @return The next property expression  
221 \*/  
222 public String next(final String expression) {  
223 if (expression == null || expression.length() == 0) {  
224 return null;  
225 }  
226 boolean indexed = false;  
227 boolean mapped = false;  
228 for (int i = 0; i < expression.length(); i++) {  
229 final char c = expression.charAt(i);  
230 if (indexed) {  
231 if (c == INDEXED\_END) {  
232 return expression.substring(0, i + 1);  
233 }  
234 } else if (mapped) {  
235 if (c == MAPPED\_END) {  
236 return expression.substring(0, i + 1);  
237 }  
238 } else {  
239 if (c == NESTED) {  
240 return expression.substring(0, i);  
241 } else if (c == MAPPED\_START) {  
242 mapped = true;  
243 } else if (c == INDEXED\_START) {  
244 indexed = true;  
245 }  
246 }  
247 }  
248 return expression;  
249 }  
250  
251 /\*\*  
252 \* Remove the last property expresson from the  
253 \* current expression.  
254 \*  
255 \* @param expression The property expression  
256 \* @return The new expression value, with first property  
257 \* expression removed - null if there are no more expressions  
258 \*/  
259 public String remove(final String expression) {  
260 if (expression == null || expression.length() == 0) {  
261 return null;  
262 }  
263 final String property = next(expression);  
264 if (expression.length() == property.length()) {  
265 return null;  
266 }  
267 int start = property.length();  
268 if (expression.charAt(start) == NESTED) {  
269 start++;  
270 }  
271 return expression.substring(start);  
272 }  
273}