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.converters;  
018  
019import java.io.IOException;  
020import java.io.StreamTokenizer;  
021import java.io.StringReader;  
022import java.lang.reflect.Array;  
023import java.util.ArrayList;  
024import java.util.Collection;  
025import java.util.Collections;  
026import java.util.Iterator;  
027import java.util.List;  
028  
029import org.apache.commons.beanutils.ConversionException;  
030import org.apache.commons.beanutils.Converter;  
031  
032/\*\*  
033 \* Generic {@link Converter} implementation that handles conversion  
034 \* to and from <b>array</b> objects.  
035 \* <p>  
036 \* Can be configured to either return a <i>default value</i> or throw a  
037 \* <code>ConversionException</code> if a conversion error occurs.  
038 \* <p>  
039 \* The main features of this implementation are:  
040 \* <ul>  
041 \* <li><b>Element Conversion</b> - delegates to a {@link Converter},  
042 \* appropriate for the type, to convert individual elements  
043 \* of the array. This leverages the power of existing converters  
044 \* without having to replicate their functionality for converting  
045 \* to the element type and removes the need to create a specifc  
046 \* array type converters.</li>  
047 \* <li><b>Arrays or Collections</b> - can convert from either arrays or  
048 \* Collections to an array, limited only by the capability  
049 \* of the delegate {@link Converter}.</li>  
050 \* <li><b>Delimited Lists</b> - can Convert <b>to</b> and <b>from</b> a  
051 \* delimited list in String format.</li>  
052 \* <li><b>Conversion to String</b> - converts an array to a  
053 \* <code>String</code> in one of two ways: as a <i>delimited list</i>  
054 \* or by converting the first element in the array to a String - this  
055 \* is controlled by the {@link ArrayConverter#setOnlyFirstToString(boolean)}  
056 \* parameter.</li>  
057 \* <li><b>Multi Dimensional Arrays</b> - it is possible to convert a <code>String</code>  
058 \* to a multi-dimensional arrays, by embedding {@link ArrayConverter}  
059 \* within each other - see example below.</li>  
060 \* <li><b>Default Value</b></li>  
061 \* <ul>  
062 \* <li><b><i>No Default</b></i> - use the  
063 \* {@link ArrayConverter#ArrayConverter(Class, Converter)}  
064 \* constructor to create a converter which throws a  
065 \* {@link ConversionException} if the value is missing or  
066 \* invalid.</li>  
067 \* <li><b><i>Default values</b></i> - use the  
068 \* {@link ArrayConverter#ArrayConverter(Class, Converter, int)}  
069 \* constructor to create a converter which returns a <i>default  
070 \* value</i>. The <i>defaultSize</i> parameter controls the  
071 \* <i>default value</i> in the following way:</li>  
072 \* <ul>  
073 \* <li><i>defaultSize < 0</i> - default is <code>null</code></li>  
074 \* <li><i>defaultSize = 0</i> - default is an array of length zero</li>  
075 \* <li><i>defaultSize > 0</i> - default is an array with a  
076 \* length specified by <code>defaultSize</code> (N.B. elements  
077 \* in the array will be <code>null</code>)</li>  
078 \* </ul>  
079 \* </ul>  
080 \* </ul>  
081 \*  
082 \* <h3>Parsing Delimited Lists</h3>  
083 \* This implementation can convert a delimited list in <code>String</code> format  
084 \* into an array of the appropriate type. By default, it uses a comma as the delimiter  
085 \* but the following methods can be used to configure parsing:  
086 \* <ul>  
087 \* <li><code>setDelimiter(char)</code> - allows the character used as  
088 \* the delimiter to be configured [default is a comma].</li>  
089 \* <li><code>setAllowedChars(char[])</code> - adds additional characters  
090 \* (to the default alphabetic/numeric) to those considered to be  
091 \* valid token characters.  
092 \* </ul>  
093 \*  
094 \* <h3>Multi Dimensional Arrays</h3>  
095 \* It is possible to convert a <code>String</code> to mulit-dimensional arrays by using  
096 \* {@link ArrayConverter} as the element {@link Converter}  
097 \* within another {@link ArrayConverter}.  
098 \* <p>  
099 \* For example, the following code demonstrates how to construct a {@link Converter}  
100 \* to convert a delimited <code>String</code> into a two dimensional integer array:  
101 \* <p>  
102 \* <pre>  
103 \* // Construct an Integer Converter  
104 \* IntegerConverter integerConverter = new IntegerConverter();  
105 \*  
106 \* // Construct an array Converter for an integer array (i.e. int[]) using  
107 \* // an IntegerConverter as the element converter.  
108 \* // N.B. Uses the default comma (i.e. ",") as the delimiter between individual numbers  
109 \* ArrayConverter arrayConverter = new ArrayConverter(int[].class, integerConverter);  
110 \*  
111 \* // Construct a "Matrix" Converter which converts arrays of integer arrays using  
112 \* // the pre-ceeding ArrayConverter as the element Converter.  
113 \* // N.B. Uses a semi-colon (i.e. ";") as the delimiter to separate the different sets of numbers.  
114 \* // Also the delimiter used by the first ArrayConverter needs to be added to the  
115 \* // "allowed characters" for this one.  
116 \* ArrayConverter matrixConverter = new ArrayConverter(int[][].class, arrayConverter);  
117 \* matrixConverter.setDelimiter(';');  
118 \* matrixConverter.setAllowedChars(new char[] {','});  
119 \*  
120 \* // Do the Conversion  
121 \* String matrixString = "11,12,13 ; 21,22,23 ; 31,32,33 ; 41,42,43";  
122 \* int[][] result = (int[][])matrixConverter.convert(int[][].class, matrixString);  
123 \* </pre>  
124 \*  
125 \* @version $Id$  
126 \* @since 1.8.0  
127 \*/  
128public class ArrayConverter extends AbstractConverter {  
129  
130 private final Class<?> defaultType;  
131 private final Converter elementConverter;  
132 private int defaultSize;  
133 private char delimiter = ',';  
134 private char[] allowedChars = new char[] {'.', '-'};  
135 private boolean onlyFirstToString = true;  
136  
137 // ----------------------------------------------------------- Constructors  
138  
139 /\*\*  
140 \* Construct an <b>array</b> <code>Converter</code> with the specified  
141 \* <b>component</b> <code>Converter</code> that throws a  
142 \* <code>ConversionException</code> if an error occurs.  
143 \*  
144 \* @param defaultType The default array type this  
145 \* <code>Converter</code> handles  
146 \* @param elementConverter Converter used to convert  
147 \* individual array elements.  
148 \*/  
149 public ArrayConverter(final Class<?> defaultType, final Converter elementConverter) {  
150 super();  
151 if (defaultType == null) {  
152 throw new IllegalArgumentException("Default type is missing");  
153 }  
154 if (!defaultType.isArray()) {  
155 throw new IllegalArgumentException("Default type must be an array.");  
156 }  
157 if (elementConverter == null) {  
158 throw new IllegalArgumentException("Component Converter is missing.");  
159 }  
160 this.defaultType = defaultType;  
161 this.elementConverter = elementConverter;  
162 }  
163  
164 /\*\*  
165 \* Construct an <b>array</b> <code>Converter</code> with the specified  
166 \* <b>component</b> <code>Converter</code> that returns a default  
167 \* array of the specified size (or <code>null</code>) if an error occurs.  
168 \*  
169 \* @param defaultType The default array type this  
170 \* <code>Converter</code> handles  
171 \* @param elementConverter Converter used to convert  
172 \* individual array elements.  
173 \* @param defaultSize Specifies the size of the default array value or if less  
174 \* than zero indicates that a <code>null</code> default value should be used.  
175 \*/  
176 public ArrayConverter(final Class<?> defaultType, final Converter elementConverter, final int defaultSize) {  
177 this(defaultType, elementConverter);  
178 this.defaultSize = defaultSize;  
179 Object defaultValue = null;  
180 if (defaultSize >= 0) {  
181 defaultValue = Array.newInstance(defaultType.getComponentType(), defaultSize);  
182 }  
183 setDefaultValue(defaultValue);  
184 }  
185  
186 /\*\*  
187 \* Set the delimiter to be used for parsing a delimited String.  
188 \*  
189 \* @param delimiter The delimiter [default ',']  
190 \*/  
191 public void setDelimiter(final char delimiter) {  
192 this.delimiter = delimiter;  
193 }  
194  
195 /\*\*  
196 \* Set the allowed characters to be used for parsing a delimited String.  
197 \*  
198 \* @param allowedChars Characters which are to be considered as part of  
199 \* the tokens when parsing a delimited String [default is '.' and '-']  
200 \*/  
201 public void setAllowedChars(final char[] allowedChars) {  
202 this.allowedChars = allowedChars;  
203 }  
204  
205 /\*\*  
206 \* Indicates whether converting to a String should create  
207 \* a delimited list or just convert the first value.  
208 \*  
209 \* @param onlyFirstToString <code>true</code> converts only  
210 \* the first value in the array to a String, <code>false</code>  
211 \* converts all values in the array into a delimited list (default  
212 \* is <code>true</code>  
213 \*/  
214 public void setOnlyFirstToString(final boolean onlyFirstToString) {  
215 this.onlyFirstToString = onlyFirstToString;  
216 }  
217  
218 /\*\*  
219 \* Return the default type this <code>Converter</code> handles.  
220 \*  
221 \* @return The default type this <code>Converter</code> handles.  
222 \*/  
223 @Override  
224 protected Class<?> getDefaultType() {  
225 return defaultType;  
226 }  
227  
228 /\*\*  
229 \* Handles conversion to a String.  
230 \*  
231 \* @param value The value to be converted.  
232 \* @return the converted String value.  
233 \* @throws Throwable if an error occurs converting to a String  
234 \*/  
235 @Override  
236 protected String convertToString(final Object value) throws Throwable {  
237  
238 int size = 0;  
239 Iterator<?> iterator = null;  
240 final Class<?> type = value.getClass();  
241 if (type.isArray()) {  
242 size = Array.getLength(value);  
243 } else {  
244 final Collection<?> collection = convertToCollection(type, value);  
245 size = collection.size();  
246 iterator = collection.iterator();  
247 }  
248  
249 if (size == 0) {  
250 return (String)getDefault(String.class);  
251 }  
252  
253 if (onlyFirstToString) {  
254 size = 1;  
255 }  
256  
257 // Create a StringBuffer containing a delimited list of the values  
258 final StringBuilder buffer = new StringBuilder();  
259 for (int i = 0; i < size; i++) {  
260 if (i > 0) {  
261 buffer.append(delimiter);  
262 }  
263 Object element = iterator == null ? Array.get(value, i) : iterator.next();  
264 element = elementConverter.convert(String.class, element);  
265 if (element != null) {  
266 buffer.append(element);  
267 }  
268 }  
269  
270 return buffer.toString();  
271  
272 }  
273  
274 /\*\*  
275 \* Handles conversion to an array of the specified type.  
276 \*  
277 \* @param <T> Target type of the conversion.  
278 \* @param type The type to which this value should be converted.  
279 \* @param value The input value to be converted.  
280 \* @return The converted value.  
281 \* @throws Throwable if an error occurs converting to the specified type  
282 \*/  
283 @Override  
284 protected <T> T convertToType(final Class<T> type, final Object value) throws Throwable {  
285  
286 if (!type.isArray()) {  
287 throw new ConversionException(toString(getClass())  
288 + " cannot handle conversion to '"  
289 + toString(type) + "' (not an array).");  
290 }  
291  
292 // Handle the source  
293 int size = 0;  
294 Iterator<?> iterator = null;  
295 if (value.getClass().isArray()) {  
296 size = Array.getLength(value);  
297 } else {  
298 final Collection<?> collection = convertToCollection(type, value);  
299 size = collection.size();  
300 iterator = collection.iterator();  
301 }  
302  
303 // Allocate a new Array  
304 final Class<?> componentType = type.getComponentType();  
305 final Object newArray = Array.newInstance(componentType, size);  
306  
307 // Convert and set each element in the new Array  
308 for (int i = 0; i < size; i++) {  
309 Object element = iterator == null ? Array.get(value, i) : iterator.next();  
310 // TODO - probably should catch conversion errors and throw  
311 // new exception providing better info back to the user  
312 element = elementConverter.convert(componentType, element);  
313 Array.set(newArray, i, element);  
314 }  
315  
316 @SuppressWarnings("unchecked")  
317 final  
318 // This is safe because T is an array type and newArray is an array of  
319 // T's component type  
320 T result = (T) newArray;  
321 return result;  
322 }  
323  
324 /\*\*  
325 \* Returns the value unchanged.  
326 \*  
327 \* @param value The value to convert  
328 \* @return The value unchanged  
329 \*/  
330 @Override  
331 protected Object convertArray(final Object value) {  
332 return value;  
333 }  
334  
335 /\*\*  
336 \* Converts non-array values to a Collection prior  
337 \* to being converted either to an array or a String.  
338 \* </p>  
339 \* <ul>  
340 \* <li>{@link Collection} values are returned unchanged</li>  
341 \* <li>{@link Number}, {@link Boolean} and {@link java.util.Date}  
342 \* values returned as a the only element in a List.</li>  
343 \* <li>All other types are converted to a String and parsed  
344 \* as a delimited list.</li>  
345 \* </ul>  
346 \*  
347 \* <strong>N.B.</strong> The method is called by both the  
348 \* {@link ArrayConverter#convertToType(Class, Object)} and  
349 \* {@link ArrayConverter#convertToString(Object)} methods for  
350 \* <i>non-array</i> types.  
351 \*  
352 \* @param type The type to convert the value to  
353 \* @param value value to be converted  
354 \* @return Collection elements.  
355 \*/  
356 protected Collection<?> convertToCollection(final Class<?> type, final Object value) {  
357 if (value instanceof Collection) {  
358 return (Collection<?>)value;  
359 }  
360 if (value instanceof Number ||  
361 value instanceof Boolean ||  
362 value instanceof java.util.Date) {  
363 final List<Object> list = new ArrayList<Object>(1);  
364 list.add(value);  
365 return list;  
366 }  
367  
368 return parseElements(type, value.toString());  
369 }  
370  
371 /\*\*  
372 \* Return the default value for conversions to the specified  
373 \* type.  
374 \* @param type Data type to which this value should be converted.  
375 \* @return The default value for the specified type.  
376 \*/  
377 @Override  
378 protected Object getDefault(final Class<?> type) {  
379 if (type.equals(String.class)) {  
380 return null;  
381 }  
382  
383 final Object defaultValue = super.getDefault(type);  
384 if (defaultValue == null) {  
385 return null;  
386 }  
387  
388 if (defaultValue.getClass().equals(type)) {  
389 return defaultValue;  
390 } else {  
391 return Array.newInstance(type.getComponentType(), defaultSize);  
392 }  
393  
394 }  
395  
396 /\*\*  
397 \* Provide a String representation of this array converter.  
398 \*  
399 \* @return A String representation of this array converter  
400 \*/  
401 @Override  
402 public String toString() {  
403 final StringBuilder buffer = new StringBuilder();  
404 buffer.append(toString(getClass()));  
405 buffer.append("[UseDefault=");  
406 buffer.append(isUseDefault());  
407 buffer.append(", ");  
408 buffer.append(elementConverter.toString());  
409 buffer.append(']');  
410 return buffer.toString();  
411 }  
412  
413 /\*\*  
414 \* <p>Parse an incoming String of the form similar to an array initializer  
415 \* in the Java language into a <code>List</code> individual Strings  
416 \* for each element, according to the following rules.</p>  
417 \* <ul>  
418 \* <li>The string is expected to be a comma-separated list of values.</li>  
419 \* <li>The string may optionally have matching '{' and '}' delimiters  
420 \* around the list.</li>  
421 \* <li>Whitespace before and after each element is stripped.</li>  
422 \* <li>Elements in the list may be delimited by single or double quotes.  
423 \* Within a quoted elements, the normal Java escape sequences are valid.</li>  
424 \* </ul>  
425 \*  
426 \* @param type The type to convert the value to  
427 \* @param value String value to be parsed  
428 \* @return List of parsed elements.  
429 \*  
430 \* @throws ConversionException if the syntax of <code>svalue</code>  
431 \* is not syntactically valid  
432 \* @throws NullPointerException if <code>svalue</code>  
433 \* is <code>null</code>  
434 \*/  
435 private List<String> parseElements(final Class<?> type, String value) {  
436  
437 if (log().isDebugEnabled()) {  
438 log().debug("Parsing elements, delimiter=[" + delimiter + "], value=[" + value + "]");  
439 }  
440  
441 // Trim any matching '{' and '}' delimiters  
442 value = value.trim();  
443 if (value.startsWith("{") && value.endsWith("}")) {  
444 value = value.substring(1, value.length() - 1);  
445 }  
446  
447 try {  
448  
449 // Set up a StreamTokenizer on the characters in this String  
450 final StreamTokenizer st = new StreamTokenizer(new StringReader(value));  
451 st.whitespaceChars(delimiter , delimiter); // Set the delimiters  
452 st.ordinaryChars('0', '9'); // Needed to turn off numeric flag  
453 st.wordChars('0', '9'); // Needed to make part of tokens  
454 for (char allowedChar : allowedChars) {  
455 st.ordinaryChars(allowedChar, allowedChar);  
456 st.wordChars(allowedChar, allowedChar);  
457 }  
458  
459 // Split comma-delimited tokens into a List  
460 List<String> list = null;  
461 while (true) {  
462 final int ttype = st.nextToken();  
463 if ((ttype == StreamTokenizer.TT\_WORD) || (ttype > 0)) {  
464 if (st.sval != null) {  
465 if (list == null) {  
466 list = new ArrayList<String>();  
467 }  
468 list.add(st.sval);  
469 }  
470 } else if (ttype == StreamTokenizer.TT\_EOF) {  
471 break;  
472 } else {  
473 throw new ConversionException("Encountered token of type "  
474 + ttype + " parsing elements to '" + toString(type) + ".");  
475 }  
476 }  
477  
478 if (list == null) {  
479 list = Collections.emptyList();  
480 }  
481 if (log().isDebugEnabled()) {  
482 log().debug(list.size() + " elements parsed");  
483 }  
484  
485 // Return the completed list  
486 return (list);  
487  
488 } catch (final IOException e) {  
489  
490 throw new ConversionException("Error converting from String to '"  
491 + toString(type) + "': " + e.getMessage(), e);  
492  
493 }  
494  
495 }  
496  
497}