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.converters;  
020  
021  
022import java.io.IOException;  
023import java.io.StreamTokenizer;  
024import java.io.StringReader;  
025import java.util.ArrayList;  
026import java.util.List;  
027import org.apache.commons.beanutils.ConversionException;  
028import org.apache.commons.beanutils.Converter;  
029  
030  
031  
032/\*\*  
033 \* <p>Convenience base class for converters that translate the String  
034 \* representation of an array into a corresponding array of primitives  
035 \* object. This class encapsulates the functionality required to parse  
036 \* the String into a list of String elements that can later be  
037 \* individually converted to the appropriate primitive type.</p>  
038 \*  
039 \* <p>The input syntax accepted by the <code>parseElements()</code> method  
040 \* is designed to be compatible with the syntax used to initialize arrays  
041 \* in a Java source program, except that only String literal values are  
042 \* supported. For maximum flexibility, the surrounding '{' and '}'  
043 \* characters are optional, and individual elements may be separated by  
044 \* any combination of whitespace and comma characters.</p>  
045 \*  
046 \* @version $Id$  
047 \* @since 1.4  
048 \* @deprecated Replaced by the new {@link ArrayConverter} implementation  
049 \*/  
050  
051@Deprecated  
052public abstract class AbstractArrayConverter implements Converter {  
053  
054  
055 // ----------------------------------------------------------- Constructors  
056  
057  
058 /\*\*  
059 \* Create a {@link Converter} that will throw a {@link ConversionException}  
060 \* if a conversion error occurs.  
061 \*/  
062 public AbstractArrayConverter() {  
063  
064 this.defaultValue = null;  
065 this.useDefault = false;  
066  
067 }  
068  
069 /\*\*  
070 \* Create a {@link Converter} that will return the specified default value  
071 \* if a conversion error occurs.  
072 \*  
073 \* @param defaultValue The default value to be returned  
074 \* @since 1.8.0  
075 \*/  
076 public AbstractArrayConverter(final Object defaultValue) {  
077  
078 if (defaultValue == NO\_DEFAULT) {  
079 this.useDefault = false;  
080 } else {  
081 this.defaultValue = defaultValue;  
082 this.useDefault = true;  
083 }  
084  
085 }  
086  
087 // ------------------------------------------------------- Static Variables  
088  
089 /\*\*  
090 \* This is a special reference that can be passed as the "default object"  
091 \* to the constructor to indicate that no default is desired. Note that  
092 \* the value 'null' cannot be used for this purpose, as the caller may  
093 \* want a null to be returned as the default.  
094 \* @since 1.8.0  
095 \*/  
096 public static final Object NO\_DEFAULT = new Object();  
097  
098 // ----------------------------------------------------- Instance Variables  
099  
100  
101 /\*\*  
102 \* <p>Model object for string arrays.</p>  
103 \*/  
104 protected static String[] strings = new String[0];  
105  
106  
107 /\*\*  
108 \* The default value specified to our Constructor, if any.  
109 \*/  
110 protected Object defaultValue = null;  
111  
112  
113 /\*\*  
114 \* Should we return the default value on conversion errors?  
115 \*/  
116 protected boolean useDefault = true;  
117  
118  
119 // --------------------------------------------------------- Public Methods  
120  
121  
122 /\*\*  
123 \* Convert the specified input object into an output object of the  
124 \* specified type. This method must be implemented by a concrete  
125 \* subclass.  
126 \*  
127 \* @param type Data type to which this value should be converted  
128 \* @param value The input value to be converted  
129 \* @return The converted value  
130 \*  
131 \* @throws ConversionException if conversion cannot be performed  
132 \* successfully  
133 \*/  
134 public abstract Object convert(Class type, Object value);  
135  
136  
137 // ------------------------------------------------------ Protected Methods  
138  
139  
140 /\*\*  
141 \* <p>Parse an incoming String of the form similar to an array initializer  
142 \* in the Java language into a <code>List</code> individual Strings  
143 \* for each element, according to the following rules.</p>  
144 \* <ul>  
145 \* <li>The string is expected to be a comma-separated list of values.</li>  
146 \* <li>The string may optionally have matching '{' and '}' delimiters  
147 \* around the list.</li>  
148 \* <li>Whitespace before and after each element is stripped.</li>  
149 \* <li>Elements in the list may be delimited by single or double quotes.  
150 \* Within a quoted elements, the normal Java escape sequences are valid.</li>  
151 \* </ul>  
152 \*  
153 \* @param svalue String value to be parsed  
154 \* @return The parsed list of String values  
155 \*  
156 \* @throws ConversionException if the syntax of <code>svalue</code>  
157 \* is not syntactically valid  
158 \* @throws NullPointerException if <code>svalue</code>  
159 \* is <code>null</code>  
160 \*/  
161 protected List parseElements(String svalue) {  
162  
163 // Validate the passed argument  
164 if (svalue == null) {  
165 throw new NullPointerException();  
166 }  
167  
168 // Trim any matching '{' and '}' delimiters  
169 svalue = svalue.trim();  
170 if (svalue.startsWith("{") && svalue.endsWith("}")) {  
171 svalue = svalue.substring(1, svalue.length() - 1);  
172 }  
173  
174 try {  
175  
176 // Set up a StreamTokenizer on the characters in this String  
177 final StreamTokenizer st =  
178 new StreamTokenizer(new StringReader(svalue));  
179 st.whitespaceChars(',',','); // Commas are delimiters  
180 st.ordinaryChars('0', '9'); // Needed to turn off numeric flag  
181 st.ordinaryChars('.', '.');  
182 st.ordinaryChars('-', '-');  
183 st.wordChars('0', '9'); // Needed to make part of tokens  
184 st.wordChars('.', '.');  
185 st.wordChars('-', '-');  
186  
187 // Split comma-delimited tokens into a List  
188 final ArrayList list = new ArrayList();  
189 while (true) {  
190 final int ttype = st.nextToken();  
191 if ((ttype == StreamTokenizer.TT\_WORD) ||  
192 (ttype > 0)) {  
193 list.add(st.sval);  
194 } else if (ttype == StreamTokenizer.TT\_EOF) {  
195 break;  
196 } else {  
197 throw new ConversionException  
198 ("Encountered token of type " + ttype);  
199 }  
200 }  
201  
202 // Return the completed list  
203 return (list);  
204  
205 } catch (final IOException e) {  
206  
207 throw new ConversionException(e);  
208  
209 }  
210  
211  
212  
213 }  
214  
215  
216}