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.lang.reflect.Array;  
020import java.util.Collection;  
021  
022import org.apache.commons.beanutils.BeanUtils;  
023import org.apache.commons.beanutils.ConversionException;  
024import org.apache.commons.beanutils.ConvertUtils;  
025import org.apache.commons.beanutils.Converter;  
026import org.apache.commons.logging.Log;  
027import org.apache.commons.logging.LogFactory;  
028  
029/\*\*  
030 \* Base {@link Converter} implementation that provides the structure  
031 \* for handling conversion <b>to</b> and <b>from</b> a specified type.  
032 \* <p>  
033 \* This implementation provides the basic structure for  
034 \* converting to/from a specified type optionally using a default  
035 \* value or throwing a {@link ConversionException} if a  
036 \* conversion error occurs.  
037 \* <p>  
038 \* Implementations should provide conversion to the specified  
039 \* type and from the specified type to a <code>String</code> value  
040 \* by implementing the following methods:  
041 \* <ul>  
042 \* <li><code>convertToString(value)</code> - convert to a String  
043 \* (default implementation uses the objects <code>toString()</code>  
044 \* method).</li>  
045 \* <li><code>convertToType(Class, value)</code> - convert  
046 \* to the specified type</li>  
047 \* </ul>  
048 \* <p>  
049 \* The default value has to be compliant to the default type of this  
050 \* converter - which is enforced by the generic type parameter. If a  
051 \* conversion is not possible and a default value is set, the converter  
052 \* tries to transform the default value to the requested target type.  
053 \* If this fails, a {@code ConversionException} if thrown.  
054 \*  
055 \* @version $Id$  
056 \* @since 1.8.0  
057 \*/  
058public abstract class AbstractConverter implements Converter {  
059  
060 /\*\* Debug logging message to indicate default value configuration \*/  
061 private static final String DEFAULT\_CONFIG\_MSG =  
062 "(N.B. Converters can be configured to use default values to avoid throwing exceptions)";  
063  
064 /\*\* Current package name \*/  
065 // getPackage() below returns null on some platforms/jvm versions during the unit tests.  
066// private static final String PACKAGE = AbstractConverter.class.getPackage().getName() + ".";  
067 private static final String PACKAGE = "org.apache.commons.beanutils.converters.";  
068  
069 /\*\*  
070 \* Logging for this instance.  
071 \*/  
072 private transient Log log;  
073  
074 /\*\*  
075 \* Should we return the default value on conversion errors?  
076 \*/  
077 private boolean useDefault = false;  
078  
079 /\*\*  
080 \* The default value specified to our Constructor, if any.  
081 \*/  
082 private Object defaultValue = null;  
083  
084 // ----------------------------------------------------------- Constructors  
085  
086 /\*\*  
087 \* Construct a <i>Converter</i> that throws a  
088 \* <code>ConversionException</code> if an error occurs.  
089 \*/  
090 public AbstractConverter() {  
091 }  
092  
093 /\*\*  
094 \* Construct a <i>Converter</i> that returns a default  
095 \* value if an error occurs.  
096 \*  
097 \* @param defaultValue The default value to be returned  
098 \* if the value to be converted is missing or an error  
099 \* occurs converting the value.  
100 \*/  
101 public AbstractConverter(final Object defaultValue) {  
102 setDefaultValue(defaultValue);  
103 }  
104  
105 // --------------------------------------------------------- Public Methods  
106  
107 /\*\*  
108 \* Indicates whether a default value will be returned or exception  
109 \* thrown in the event of a conversion error.  
110 \*  
111 \* @return <code>true</code> if a default value will be returned for  
112 \* conversion errors or <code>false</code> if a {@link ConversionException}  
113 \* will be thrown.  
114 \*/  
115 public boolean isUseDefault() {  
116 return useDefault;  
117 }  
118  
119 /\*\*  
120 \* Convert the input object into an output object of the  
121 \* specified type.  
122 \*  
123 \* @param <T> the target type of the conversion  
124 \* @param type Data type to which this value should be converted  
125 \* @param value The input value to be converted  
126 \* @return The converted value.  
127 \* @throws ConversionException if conversion cannot be performed  
128 \* successfully and no default is specified.  
129 \*/  
130 public <T> T convert(final Class<T> type, Object value) {  
131  
132 if (type == null) {  
133 return convertToDefaultType(type, value);  
134 }  
135  
136 Class<?> sourceType = value == null ? null : value.getClass();  
137 final Class<T> targetType = ConvertUtils.primitiveToWrapper(type);  
138  
139 if (log().isDebugEnabled()) {  
140 log().debug("Converting"  
141 + (value == null ? "" : " '" + toString(sourceType) + "'")  
142 + " value '" + value + "' to type '" + toString(targetType) + "'");  
143 }  
144  
145 value = convertArray(value);  
146  
147 // Missing Value  
148 if (value == null) {  
149 return handleMissing(targetType);  
150 }  
151  
152 sourceType = value.getClass();  
153  
154 try {  
155 // Convert --> String  
156 if (targetType.equals(String.class)) {  
157 return targetType.cast(convertToString(value));  
158  
159 // No conversion necessary  
160 } else if (targetType.equals(sourceType)) {  
161 if (log().isDebugEnabled()) {  
162 log().debug(" No conversion required, value is already a "  
163 + toString(targetType));  
164 }  
165 return targetType.cast(value);  
166  
167 // Convert --> Type  
168 } else {  
169 final Object result = convertToType(targetType, value);  
170 if (log().isDebugEnabled()) {  
171 log().debug(" Converted to " + toString(targetType) +  
172 " value '" + result + "'");  
173 }  
174 return targetType.cast(result);  
175 }  
176 } catch (final Throwable t) {  
177 return handleError(targetType, value, t);  
178 }  
179  
180 }  
181  
182 /\*\*  
183 \* Convert the input object into a String.  
184 \* <p>  
185 \* <b>N.B.</b>This implementation simply uses the value's  
186 \* <code>toString()</code> method and should be overriden if a  
187 \* more sophisticated mechanism for <i>conversion to a String</i>  
188 \* is required.  
189 \*  
190 \* @param value The input value to be converted.  
191 \* @return the converted String value.  
192 \* @throws Throwable if an error occurs converting to a String  
193 \*/  
194 protected String convertToString(final Object value) throws Throwable {  
195 return value.toString();  
196 }  
197  
198 /\*\*  
199 \* Convert the input object into an output object of the  
200 \* specified type.  
201 \* <p>  
202 \* Typical implementations will provide a minimum of  
203 \* <code>String --> type</code> conversion.  
204 \*  
205 \* @param <T> Target type of the conversion.  
206 \* @param type Data type to which this value should be converted.  
207 \* @param value The input value to be converted.  
208 \* @return The converted value.  
209 \* @throws Throwable if an error occurs converting to the specified type  
210 \*/  
211 protected abstract <T> T convertToType(Class<T> type, Object value) throws Throwable;  
212  
213 /\*\*  
214 \* Return the first element from an Array (or Collection)  
215 \* or the value unchanged if not an Array (or Collection).  
216 \*  
217 \* N.B. This needs to be overriden for array/Collection converters.  
218 \*  
219 \* @param value The value to convert  
220 \* @return The first element in an Array (or Collection)  
221 \* or the value unchanged if not an Array (or Collection)  
222 \*/  
223 protected Object convertArray(final Object value) {  
224 if (value == null) {  
225 return null;  
226 }  
227 if (value.getClass().isArray()) {  
228 if (Array.getLength(value) > 0) {  
229 return Array.get(value, 0);  
230 } else {  
231 return null;  
232 }  
233 }  
234 if (value instanceof Collection) {  
235 final Collection<?> collection = (Collection<?>)value;  
236 if (collection.size() > 0) {  
237 return collection.iterator().next();  
238 } else {  
239 return null;  
240 }  
241 }  
242 return value;  
243 }  
244  
245 /\*\*  
246 \* Handle Conversion Errors.  
247 \* <p>  
248 \* If a default value has been specified then it is returned  
249 \* otherwise a ConversionException is thrown.  
250 \*  
251 \* @param <T> Target type of the conversion.  
252 \* @param type Data type to which this value should be converted.  
253 \* @param value The input value to be converted  
254 \* @param cause The exception thrown by the <code>convert</code> method  
255 \* @return The default value.  
256 \* @throws ConversionException if no default value has been  
257 \* specified for this {@link Converter}.  
258 \*/  
259 protected <T> T handleError(final Class<T> type, final Object value, final Throwable cause) {  
260 if (log().isDebugEnabled()) {  
261 if (cause instanceof ConversionException) {  
262 log().debug(" Conversion threw ConversionException: " + cause.getMessage());  
263 } else {  
264 log().debug(" Conversion threw " + cause);  
265 }  
266 }  
267  
268 if (useDefault) {  
269 return handleMissing(type);  
270 }  
271  
272 ConversionException cex = null;  
273 if (cause instanceof ConversionException) {  
274 cex = (ConversionException)cause;  
275 if (log().isDebugEnabled()) {  
276 log().debug(" Re-throwing ConversionException: " + cex.getMessage());  
277 log().debug(" " + DEFAULT\_CONFIG\_MSG);  
278 }  
279 } else {  
280 final String msg = "Error converting from '" + toString(value.getClass()) +  
281 "' to '" + toString(type) + "' " + cause.getMessage();  
282 cex = new ConversionException(msg, cause);  
283 if (log().isDebugEnabled()) {  
284 log().debug(" Throwing ConversionException: " + msg);  
285 log().debug(" " + DEFAULT\_CONFIG\_MSG);  
286 }  
287 BeanUtils.initCause(cex, cause);  
288 }  
289  
290 throw cex;  
291  
292 }  
293  
294 /\*\*  
295 \* Handle missing values.  
296 \* <p>  
297 \* If a default value has been specified, then it is returned (after a cast  
298 \* to the desired target class); otherwise a ConversionException is thrown.  
299 \*  
300 \* @param <T> the desired target type  
301 \* @param type Data type to which this value should be converted.  
302 \* @return The default value.  
303 \* @throws ConversionException if no default value has been  
304 \* specified for this {@link Converter}.  
305 \*/  
306 protected <T> T handleMissing(final Class<T> type) {  
307  
308 if (useDefault || type.equals(String.class)) {  
309 Object value = getDefault(type);  
310 if (useDefault && value != null && !(type.equals(value.getClass()))) {  
311 try {  
312 value = convertToType(type, defaultValue);  
313 } catch (final Throwable t) {  
314 throw new ConversionException("Default conversion to " + toString(type)  
315 + " failed.", t);  
316 }  
317 }  
318 if (log().isDebugEnabled()) {  
319 log().debug(" Using default "  
320 + (value == null ? "" : toString(value.getClass()) + " ")  
321 + "value '" + defaultValue + "'");  
322 }  
323 // value is now either null or of the desired target type  
324 return type.cast(value);  
325 }  
326  
327 final ConversionException cex = new ConversionException("No value specified for '" +  
328 toString(type) + "'");  
329 if (log().isDebugEnabled()) {  
330 log().debug(" Throwing ConversionException: " + cex.getMessage());  
331 log().debug(" " + DEFAULT\_CONFIG\_MSG);  
332 }  
333 throw cex;  
334  
335 }  
336  
337 /\*\*  
338 \* Set the default value, converting as required.  
339 \* <p>  
340 \* If the default value is different from the type the  
341 \* <code>Converter</code> handles, it will be converted  
342 \* to the handled type.  
343 \*  
344 \* @param defaultValue The default value to be returned  
345 \* if the value to be converted is missing or an error  
346 \* occurs converting the value.  
347 \* @throws ConversionException if an error occurs converting  
348 \* the default value  
349 \*/  
350 protected void setDefaultValue(final Object defaultValue) {  
351 useDefault = false;  
352 if (log().isDebugEnabled()) {  
353 log().debug("Setting default value: " + defaultValue);  
354 }  
355 if (defaultValue == null) {  
356 this.defaultValue = null;  
357 } else {  
358 this.defaultValue = convert(getDefaultType(), defaultValue);  
359 }  
360 useDefault = true;  
361 }  
362  
363 /\*\*  
364 \* Return the default type this <code>Converter</code> handles.  
365 \*  
366 \* @return The default type this <code>Converter</code> handles.  
367 \*/  
368 protected abstract Class<?> getDefaultType();  
369  
370 /\*\*  
371 \* Return the default value for conversions to the specified  
372 \* type.  
373 \* @param type Data type to which this value should be converted.  
374 \* @return The default value for the specified type.  
375 \*/  
376 protected Object getDefault(final Class<?> type) {  
377 if (type.equals(String.class)) {  
378 return null;  
379 } else {  
380 return defaultValue;  
381 }  
382 }  
383  
384 /\*\*  
385 \* Provide a String representation of this converter.  
386 \*  
387 \* @return A String representation of this converter  
388 \*/  
389 @Override  
390 public String toString() {  
391 return toString(getClass()) + "[UseDefault=" + useDefault + "]";  
392 }  
393  
394 // ----------------------------------------------------------- Package Methods  
395  
396 /\*\*  
397 \* Accessor method for Log instance.  
398 \* <p>  
399 \* The Log instance variable is transient and  
400 \* accessing it through this method ensures it  
401 \* is re-initialized when this instance is  
402 \* de-serialized.  
403 \*  
404 \* @return The Log instance.  
405 \*/  
406 Log log() {  
407 if (log == null) {  
408 log = LogFactory.getLog(getClass());  
409 }  
410 return log;  
411 }  
412  
413 /\*\*  
414 \* Provide a String representation of a <code>java.lang.Class</code>.  
415 \* @param type The <code>java.lang.Class</code>.  
416 \* @return The String representation.  
417 \*/  
418 String toString(final Class<?> type) {  
419 String typeName = null;  
420 if (type == null) {  
421 typeName = "null";  
422 } else if (type.isArray()) {  
423 Class<?> elementType = type.getComponentType();  
424 int count = 1;  
425 while (elementType.isArray()) {  
426 elementType = elementType .getComponentType();  
427 count++;  
428 }  
429 typeName = elementType.getName();  
430 for (int i = 0; i < count; i++) {  
431 typeName += "[]";  
432 }  
433 } else {  
434 typeName = type.getName();  
435 }  
436 if (typeName.startsWith("java.lang.") ||  
437 typeName.startsWith("java.util.") ||  
438 typeName.startsWith("java.math.")) {  
439 typeName = typeName.substring("java.lang.".length());  
440 } else if (typeName.startsWith(PACKAGE)) {  
441 typeName = typeName.substring(PACKAGE.length());  
442 }  
443 return typeName;  
444 }  
445  
446 /\*\*  
447 \* Performs a conversion to the default type. This method is called if we do  
448 \* not have a target class. In this case, the T parameter is not set.  
449 \* Therefore, we can cast to it (which is required to fulfill the contract  
450 \* of the method signature).  
451 \*  
452 \* @param <T> the type of the result object  
453 \* @param targetClass the target class of the conversion  
454 \* @param value the value to be converted  
455 \* @return the converted value  
456 \*/  
457 private <T> T convertToDefaultType(final Class<T> targetClass, final Object value) {  
458 @SuppressWarnings("unchecked")  
459 final  
460 T result = (T) convert(getDefaultType(), value);  
461 return result;  
462 }  
463  
464 /\*\*  
465 \* Generates a standard conversion exception with a message indicating that  
466 \* the passed in value cannot be converted to the desired target type.  
467 \*  
468 \* @param type the target type  
469 \* @param value the value to be converted  
470 \* @return a {@code ConversionException} with a standard message  
471 \* @since 1.9  
472 \*/  
473 protected ConversionException conversionException(final Class<?> type, final Object value) {  
474 return new ConversionException("Can't convert value '" + value  
475 + "' to type " + type);  
476 }  
477}