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.math.BigDecimal;  
020import java.math.BigInteger;  
021import java.text.DecimalFormat;  
022import java.text.DecimalFormatSymbols;  
023import java.text.NumberFormat;  
024import java.text.ParsePosition;  
025import java.util.Calendar;  
026import java.util.Date;  
027import java.util.Locale;  
028  
029import org.apache.commons.beanutils.ConversionException;  
030  
031/\*\*  
032 \* {@link org.apache.commons.beanutils.Converter} implementaion that handles conversion  
033 \* to and from <b>java.lang.Number</b> objects.  
034 \* <p>  
035 \* This implementation handles conversion for the following  
036 \* <code>java.lang.Number</code> types.  
037 \* <ul>  
038 \* <li><code>java.lang.Byte</code></li>  
039 \* <li><code>java.lang.Short</code></li>  
040 \* <li><code>java.lang.Integer</code></li>  
041 \* <li><code>java.lang.Long</code></li>  
042 \* <li><code>java.lang.Float</code></li>  
043 \* <li><code>java.lang.Double</code></li>  
044 \* <li><code>java.math.BigDecimal</code></li>  
045 \* <li><code>java.math.BigInteger</code></li>  
046 \* </ul>  
047 \*  
048 \* <h3>String Conversions (to and from)</h3>  
049 \* This class provides a number of ways in which number  
050 \* conversions to/from Strings can be achieved:  
051 \* <ul>  
052 \* <li>Using the default format for the default Locale, configure using:</li>  
053 \* <ul>  
054 \* <li><code>setUseLocaleFormat(true)</code></li>  
055 \* </ul>  
056 \* <li>Using the default format for a specified Locale, configure using:</li>  
057 \* <ul>  
058 \* <li><code>setLocale(Locale)</code></li>  
059 \* </ul>  
060 \* <li>Using a specified pattern for the default Locale, configure using:</li>  
061 \* <ul>  
062 \* <li><code>setPattern(String)</code></li>  
063 \* </ul>  
064 \* <li>Using a specified pattern for a specified Locale, configure using:</li>  
065 \* <ul>  
066 \* <li><code>setPattern(String)</code></li>  
067 \* <li><code>setLocale(Locale)</code></li>  
068 \* </ul>  
069 \* <li>If none of the above are configured the  
070 \* <code>toNumber(String)</code> method is used to convert  
071 \* from String to Number and the Number's  
072 \* <code>toString()</code> method used to convert from  
073 \* Number to String.</li>  
074 \* </ul>  
075 \*  
076 \* <p>  
077 \* <strong>N.B.</strong>Patterns can only be specified using the <i>standard</i>  
078 \* pattern characters and NOT in <i>localized</i> form (see <code>java.text.DecimalFormat</code>).  
079 \* For example to cater for number styles used in Germany such as <code>0.000,00</code> the pattern  
080 \* is specified in the normal form <code>0,000.00</code> and the locale set to <code>Locale.GERMANY</code>.  
081 \*  
082 \* @version $Id$  
083 \* @since 1.8.0  
084 \*/  
085public abstract class NumberConverter extends AbstractConverter {  
086  
087 private static final Integer ZERO = new Integer(0);  
088 private static final Integer ONE = new Integer(1);  
089  
090 private String pattern;  
091 private final boolean allowDecimals;  
092 private boolean useLocaleFormat;  
093 private Locale locale;  
094  
095 // ----------------------------------------------------------- Constructors  
096  
097 /\*\*  
098 \* Construct a <b>java.lang.Number</b> <i>Converter</i>  
099 \* that throws a <code>ConversionException</code> if a error occurs.  
100 \*  
101 \* @param allowDecimals Indicates whether decimals are allowed  
102 \*/  
103 public NumberConverter(final boolean allowDecimals) {  
104 super();  
105 this.allowDecimals = allowDecimals;  
106 }  
107  
108 /\*\*  
109 \* Construct a <code>java.lang.Number</code> <i>Converter</i> that returns  
110 \* a default value if an error occurs.  
111 \*  
112 \* @param allowDecimals Indicates whether decimals are allowed  
113 \* @param defaultValue The default value to be returned  
114 \*/  
115 public NumberConverter(final boolean allowDecimals, final Object defaultValue) {  
116 super();  
117 this.allowDecimals = allowDecimals;  
118 setDefaultValue(defaultValue);  
119 }  
120  
121 // --------------------------------------------------------- Public Methods  
122  
123 /\*\*  
124 \* Return whether decimals are allowed in the number.  
125 \*  
126 \* @return Whether decimals are allowed in the number  
127 \*/  
128 public boolean isAllowDecimals() {  
129 return allowDecimals;  
130 }  
131  
132 /\*\*  
133 \* Set whether a format should be used to convert  
134 \* the Number.  
135 \*  
136 \* @param useLocaleFormat <code>true</code> if a number format  
137 \* should be used.  
138 \*/  
139 public void setUseLocaleFormat(final boolean useLocaleFormat) {  
140 this.useLocaleFormat = useLocaleFormat;  
141 }  
142  
143 /\*\*  
144 \* Return the number format pattern used to convert  
145 \* Numbers to/from a <code>java.lang.String</code>  
146 \* (or <code>null</code> if none specified).  
147 \* <p>  
148 \* See <code>java.text.DecimalFormat</code> for details  
149 \* of how to specify the pattern.  
150 \*  
151 \* @return The format pattern.  
152 \*/  
153 public String getPattern() {  
154 return pattern;  
155 }  
156  
157 /\*\*  
158 \* Set a number format pattern to use to convert  
159 \* Numbers to/from a <code>java.lang.String</code>.  
160 \* <p>  
161 \* See <code>java.text.DecimalFormat</code> for details  
162 \* of how to specify the pattern.  
163 \*  
164 \* @param pattern The format pattern.  
165 \*/  
166 public void setPattern(final String pattern) {  
167 this.pattern = pattern;  
168 setUseLocaleFormat(true);  
169 }  
170  
171 /\*\*  
172 \* Return the Locale for the <i>Converter</i>  
173 \* (or <code>null</code> if none specified).  
174 \*  
175 \* @return The locale to use for conversion  
176 \*/  
177 public Locale getLocale() {  
178 return locale;  
179 }  
180  
181 /\*\*  
182 \* Set the Locale for the <i>Converter</i>.  
183 \*  
184 \* @param locale The locale to use for conversion  
185 \*/  
186 public void setLocale(final Locale locale) {  
187 this.locale = locale;  
188 setUseLocaleFormat(true);  
189 }  
190  
191 // ------------------------------------------------------ Protected Methods  
192  
193 /\*\*  
194 \* Convert an input Number object into a String.  
195 \*  
196 \* @param value The input value to be converted  
197 \* @return the converted String value.  
198 \* @throws Throwable if an error occurs converting to a String  
199 \*/  
200 @Override  
201 protected String convertToString(final Object value) throws Throwable {  
202  
203 String result = null;  
204 if (useLocaleFormat && value instanceof Number) {  
205 final NumberFormat format = getFormat();  
206 format.setGroupingUsed(false);  
207 result = format.format(value);  
208 if (log().isDebugEnabled()) {  
209 log().debug(" Converted to String using format '" + result + "'");  
210 }  
211  
212 } else {  
213 result = value.toString();  
214 if (log().isDebugEnabled()) {  
215 log().debug(" Converted to String using toString() '" + result + "'");  
216 }  
217 }  
218 return result;  
219  
220 }  
221  
222 /\*\*  
223 \* Convert the input object into a Number object of the  
224 \* specified type.  
225 \*  
226 \* @param <T> Target type of the conversion.  
227 \* @param targetType Data type to which this value should be converted.  
228 \* @param value The input value to be converted.  
229 \* @return The converted value.  
230 \* @throws Throwable if an error occurs converting to the specified type  
231 \*/  
232 @Override  
233 protected <T> T convertToType(final Class<T> targetType, final Object value) throws Throwable {  
234  
235 final Class<?> sourceType = value.getClass();  
236 // Handle Number  
237 if (value instanceof Number) {  
238 return toNumber(sourceType, targetType, (Number)value);  
239 }  
240  
241 // Handle Boolean  
242 if (value instanceof Boolean) {  
243 return toNumber(sourceType, targetType, ((Boolean)value).booleanValue() ? ONE : ZERO);  
244 }  
245  
246 // Handle Date --> Long  
247 if (value instanceof Date && Long.class.equals(targetType)) {  
248 return targetType.cast(new Long(((Date)value).getTime()));  
249 }  
250  
251 // Handle Calendar --> Long  
252 if (value instanceof Calendar && Long.class.equals(targetType)) {  
253 return targetType.cast(new Long(((Calendar)value).getTime().getTime()));  
254 }  
255  
256 // Convert all other types to String & handle  
257 final String stringValue = value.toString().trim();  
258 if (stringValue.length() == 0) {  
259 return handleMissing(targetType);  
260 }  
261  
262 // Convert/Parse a String  
263 Number number = null;  
264 if (useLocaleFormat) {  
265 final NumberFormat format = getFormat();  
266 number = parse(sourceType, targetType, stringValue, format);  
267 } else {  
268 if (log().isDebugEnabled()) {  
269 log().debug(" No NumberFormat, using default conversion");  
270 }  
271 number = toNumber(sourceType, targetType, stringValue);  
272 }  
273  
274 // Ensure the correct number type is returned  
275 return toNumber(sourceType, targetType, number);  
276  
277 }  
278  
279 /\*\*  
280 \* Convert any Number object to the specified type for this  
281 \* <i>Converter</i>.  
282 \* <p>  
283 \* This method handles conversion to the following types:  
284 \* <ul>  
285 \* <li><code>java.lang.Byte</code></li>  
286 \* <li><code>java.lang.Short</code></li>  
287 \* <li><code>java.lang.Integer</code></li>  
288 \* <li><code>java.lang.Long</code></li>  
289 \* <li><code>java.lang.Float</code></li>  
290 \* <li><code>java.lang.Double</code></li>  
291 \* <li><code>java.math.BigDecimal</code></li>  
292 \* <li><code>java.math.BigInteger</code></li>  
293 \* </ul>  
294 \* @param sourceType The type being converted from  
295 \* @param targetType The Number type to convert to  
296 \* @param value The Number to convert.  
297 \*  
298 \* @return The converted value.  
299 \*/  
300 private <T> T toNumber(final Class<?> sourceType, final Class<T> targetType, final Number value) {  
301  
302 // Correct Number type already  
303 if (targetType.equals(value.getClass())) {  
304 return targetType.cast(value);  
305 }  
306  
307 // Byte  
308 if (targetType.equals(Byte.class)) {  
309 final long longValue = value.longValue();  
310 if (longValue > Byte.MAX\_VALUE) {  
311 throw new ConversionException(toString(sourceType) + " value '" + value  
312 + "' is too large for " + toString(targetType));  
313 }  
314 if (longValue < Byte.MIN\_VALUE) {  
315 throw new ConversionException(toString(sourceType) + " value '" + value  
316 + "' is too small " + toString(targetType));  
317 }  
318 return targetType.cast(new Byte(value.byteValue()));  
319 }  
320  
321 // Short  
322 if (targetType.equals(Short.class)) {  
323 final long longValue = value.longValue();  
324 if (longValue > Short.MAX\_VALUE) {  
325 throw new ConversionException(toString(sourceType) + " value '" + value  
326 + "' is too large for " + toString(targetType));  
327 }  
328 if (longValue < Short.MIN\_VALUE) {  
329 throw new ConversionException(toString(sourceType) + " value '" + value  
330 + "' is too small " + toString(targetType));  
331 }  
332 return targetType.cast(new Short(value.shortValue()));  
333 }  
334  
335 // Integer  
336 if (targetType.equals(Integer.class)) {  
337 final long longValue = value.longValue();  
338 if (longValue > Integer.MAX\_VALUE) {  
339 throw new ConversionException(toString(sourceType) + " value '" + value  
340 + "' is too large for " + toString(targetType));  
341 }  
342 if (longValue < Integer.MIN\_VALUE) {  
343 throw new ConversionException(toString(sourceType) + " value '" + value  
344 + "' is too small " + toString(targetType));  
345 }  
346 return targetType.cast(new Integer(value.intValue()));  
347 }  
348  
349 // Long  
350 if (targetType.equals(Long.class)) {  
351 return targetType.cast(new Long(value.longValue()));  
352 }  
353  
354 // Float  
355 if (targetType.equals(Float.class)) {  
356 if (value.doubleValue() > Float.MAX\_VALUE) {  
357 throw new ConversionException(toString(sourceType) + " value '" + value  
358 + "' is too large for " + toString(targetType));  
359 }  
360 return targetType.cast(new Float(value.floatValue()));  
361 }  
362  
363 // Double  
364 if (targetType.equals(Double.class)) {  
365 return targetType.cast(new Double(value.doubleValue()));  
366 }  
367  
368 // BigDecimal  
369 if (targetType.equals(BigDecimal.class)) {  
370 if (value instanceof Float || value instanceof Double) {  
371 return targetType.cast(new BigDecimal(value.toString()));  
372 } else if (value instanceof BigInteger) {  
373 return targetType.cast(new BigDecimal((BigInteger)value));  
374 } else if (value instanceof BigDecimal) {  
375 return targetType.cast(new BigDecimal(value.toString()));  
376 } else {  
377 return targetType.cast(BigDecimal.valueOf(value.longValue()));  
378 }  
379 }  
380  
381 // BigInteger  
382 if (targetType.equals(BigInteger.class)) {  
383 if (value instanceof BigDecimal) {  
384 return targetType.cast(((BigDecimal)value).toBigInteger());  
385 } else {  
386 return targetType.cast(BigInteger.valueOf(value.longValue()));  
387 }  
388 }  
389  
390 final String msg = toString(getClass()) + " cannot handle conversion to '"  
391 + toString(targetType) + "'";  
392 if (log().isWarnEnabled()) {  
393 log().warn(" " + msg);  
394 }  
395 throw new ConversionException(msg);  
396  
397 }  
398  
399 /\*\*  
400 \* Default String to Number conversion.  
401 \* <p>  
402 \* This method handles conversion from a String to the following types:  
403 \* <ul>  
404 \* <li><code>java.lang.Byte</code></li>  
405 \* <li><code>java.lang.Short</code></li>  
406 \* <li><code>java.lang.Integer</code></li>  
407 \* <li><code>java.lang.Long</code></li>  
408 \* <li><code>java.lang.Float</code></li>  
409 \* <li><code>java.lang.Double</code></li>  
410 \* <li><code>java.math.BigDecimal</code></li>  
411 \* <li><code>java.math.BigInteger</code></li>  
412 \* </ul>  
413 \* @param sourceType The type being converted from  
414 \* @param targetType The Number type to convert to  
415 \* @param value The String value to convert.  
416 \*  
417 \* @return The converted Number value.  
418 \*/  
419 private Number toNumber(final Class<?> sourceType, final Class<?> targetType, final String value) {  
420  
421 // Byte  
422 if (targetType.equals(Byte.class)) {  
423 return new Byte(value);  
424 }  
425  
426 // Short  
427 if (targetType.equals(Short.class)) {  
428 return new Short(value);  
429 }  
430  
431 // Integer  
432 if (targetType.equals(Integer.class)) {  
433 return new Integer(value);  
434 }  
435  
436 // Long  
437 if (targetType.equals(Long.class)) {  
438 return new Long(value);  
439 }  
440  
441 // Float  
442 if (targetType.equals(Float.class)) {  
443 return new Float(value);  
444 }  
445  
446 // Double  
447 if (targetType.equals(Double.class)) {  
448 return new Double(value);  
449 }  
450  
451 // BigDecimal  
452 if (targetType.equals(BigDecimal.class)) {  
453 return new BigDecimal(value);  
454 }  
455  
456 // BigInteger  
457 if (targetType.equals(BigInteger.class)) {  
458 return new BigInteger(value);  
459 }  
460  
461 final String msg = toString(getClass()) + " cannot handle conversion from '" +  
462 toString(sourceType) + "' to '" + toString(targetType) + "'";  
463 if (log().isWarnEnabled()) {  
464 log().warn(" " + msg);  
465 }  
466 throw new ConversionException(msg);  
467 }  
468  
469 /\*\*  
470 \* Provide a String representation of this number converter.  
471 \*  
472 \* @return A String representation of this number converter  
473 \*/  
474 @Override  
475 public String toString() {  
476 final StringBuilder buffer = new StringBuilder();  
477 buffer.append(toString(getClass()));  
478 buffer.append("[UseDefault=");  
479 buffer.append(isUseDefault());  
480 buffer.append(", UseLocaleFormat=");  
481 buffer.append(useLocaleFormat);  
482 if (pattern != null) {  
483 buffer.append(", Pattern=");  
484 buffer.append(pattern);  
485 }  
486 if (locale != null) {  
487 buffer.append(", Locale=");  
488 buffer.append(locale);  
489 }  
490 buffer.append(']');  
491 return buffer.toString();  
492 }  
493  
494 /\*\*  
495 \* Return a NumberFormat to use for Conversion.  
496 \*  
497 \* @return The NumberFormat.  
498 \*/  
499 private NumberFormat getFormat() {  
500 NumberFormat format = null;  
501 if (pattern != null) {  
502 if (locale == null) {  
503 if (log().isDebugEnabled()) {  
504 log().debug(" Using pattern '" + pattern + "'");  
505 }  
506 format = new DecimalFormat(pattern);  
507 } else {  
508 if (log().isDebugEnabled()) {  
509 log().debug(" Using pattern '" + pattern + "'" +  
510 " with Locale[" + locale + "]");  
511 }  
512 final DecimalFormatSymbols symbols = new DecimalFormatSymbols(locale);  
513 format = new DecimalFormat(pattern, symbols);  
514 }  
515 } else {  
516 if (locale == null) {  
517 if (log().isDebugEnabled()) {  
518 log().debug(" Using default Locale format");  
519 }  
520 format = NumberFormat.getInstance();  
521 } else {  
522 if (log().isDebugEnabled()) {  
523 log().debug(" Using Locale[" + locale + "] format");  
524 }  
525 format = NumberFormat.getInstance(locale);  
526 }  
527 }  
528 if (!allowDecimals) {  
529 format.setParseIntegerOnly(true);  
530 }  
531 return format;  
532 }  
533  
534 /\*\*  
535 \* Convert a String into a <code>Number</code> object.  
536 \* @param sourceType the source type of the conversion  
537 \* @param targetType The type to convert the value to  
538 \* @param value The String date value.  
539 \* @param format The NumberFormat to parse the String value.  
540 \*  
541 \* @return The converted Number object.  
542 \* @throws ConversionException if the String cannot be converted.  
543 \*/  
544 private Number parse(final Class<?> sourceType, final Class<?> targetType, final String value, final NumberFormat format) {  
545 final ParsePosition pos = new ParsePosition(0);  
546 final Number parsedNumber = format.parse(value, pos);  
547 if (pos.getErrorIndex() >= 0 || pos.getIndex() != value.length() || parsedNumber == null) {  
548 String msg = "Error converting from '" + toString(sourceType) + "' to '" + toString(targetType) + "'";  
549 if (format instanceof DecimalFormat) {  
550 msg += " using pattern '" + ((DecimalFormat)format).toPattern() + "'";  
551 }  
552 if (locale != null) {  
553 msg += " for locale=[" + locale + "]";  
554 }  
555 if (log().isDebugEnabled()) {  
556 log().debug(" " + msg);  
557 }  
558 throw new ConversionException(msg);  
559 }  
560 return parsedNumber;  
561 }  
562  
563}