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  
018package org.apache.commons.beanutils.locale;  
019  
020import java.text.ParseException;  
021import java.util.Locale;  
022  
023import org.apache.commons.beanutils.ConversionException;  
024import org.apache.commons.beanutils.ConvertUtils;  
025import org.apache.commons.logging.Log;  
026import org.apache.commons.logging.LogFactory;  
027  
028  
029/\*\*  
030 \* <p>The base class for all standart type locale-sensitive converters.  
031 \* It has {@link LocaleConverter} and {@link org.apache.commons.beanutils.Converter} implementations,  
032 \* that convert an incoming locale-sensitive Object into an object of correspond type,  
033 \* optionally using a default value or throwing a {@link ConversionException}  
034 \* if a conversion error occurs.</p>  
035 \*  
036 \* @version $Id$  
037 \*/  
038  
039public abstract class BaseLocaleConverter implements LocaleConverter {  
040  
041 // ----------------------------------------------------- Instance Variables  
042  
043 /\*\* All logging goes through this logger \*/  
044 private final Log log = LogFactory.getLog(BaseLocaleConverter.class);  
045  
046 /\*\* The default value specified to our Constructor, if any. \*/  
047 private Object defaultValue = null;  
048  
049 /\*\* Should we return the default value on conversion errors? \*/  
050 protected boolean useDefault = false;  
051  
052 /\*\* The locale specified to our Constructor, by default - system locale. \*/  
053 protected Locale locale = Locale.getDefault();  
054  
055 /\*\* The default pattern specified to our Constructor, if any. \*/  
056 protected String pattern = null;  
057  
058 /\*\* The flag indicating whether the given pattern string is localized or not. \*/  
059 protected boolean locPattern = false;  
060  
061 // ----------------------------------------------------------- Constructors  
062  
063 /\*\*  
064 \* Create a {@link LocaleConverter} that will throw a {@link ConversionException}  
065 \* if a conversion error occurs.  
066 \* An unlocalized pattern is used for the convertion.  
067 \*  
068 \* @param locale The locale  
069 \* @param pattern The convertion pattern  
070 \*/  
071 protected BaseLocaleConverter(final Locale locale, final String pattern) {  
072  
073 this(null, locale, pattern, false, false);  
074 }  
075  
076 /\*\*  
077 \* Create a {@link LocaleConverter} that will throw a {@link ConversionException}  
078 \* if a conversion error occurs.  
079 \*  
080 \* @param locale The locale  
081 \* @param pattern The convertion pattern  
082 \* @param locPattern Indicate whether the pattern is localized or not  
083 \*/  
084 protected BaseLocaleConverter(final Locale locale, final String pattern, final boolean locPattern) {  
085  
086 this(null, locale, pattern, false, locPattern);  
087 }  
088  
089 /\*\*  
090 \* Create a {@link LocaleConverter} that will return the specified default value  
091 \* if a conversion error occurs.  
092 \* An unlocalized pattern is used for the convertion.  
093 \*  
094 \* @param defaultValue The default value to be returned  
095 \* @param locale The locale  
096 \* @param pattern The convertion pattern  
097 \*/  
098 protected BaseLocaleConverter(final Object defaultValue, final Locale locale, final String pattern) {  
099  
100 this(defaultValue, locale, pattern, false);  
101 }  
102  
103 /\*\*  
104 \* Create a {@link LocaleConverter} that will return the specified default value  
105 \* if a conversion error occurs.  
106 \*  
107 \* @param defaultValue The default value to be returned  
108 \* @param locale The locale  
109 \* @param pattern The convertion pattern  
110 \* @param locPattern Indicate whether the pattern is localized or not  
111 \*/  
112 protected BaseLocaleConverter(final Object defaultValue, final Locale locale, final String pattern, final boolean locPattern) {  
113  
114 this(defaultValue, locale, pattern, true, locPattern);  
115 }  
116  
117 /\*\*  
118 \* Create a {@link LocaleConverter} that will return the specified default value  
119 \* or throw a {@link ConversionException} if a conversion error occurs.  
120 \*  
121 \* @param defaultValue The default value to be returned  
122 \* @param locale The locale  
123 \* @param pattern The convertion pattern  
124 \* @param useDefault Indicate whether the default value is used or not  
125 \* @param locPattern Indicate whether the pattern is localized or not  
126 \*/  
127 private BaseLocaleConverter(final Object defaultValue, final Locale locale,  
128 final String pattern, final boolean useDefault, final boolean locPattern) {  
129  
130 if (useDefault) {  
131 this.defaultValue = defaultValue;  
132 this.useDefault = true;  
133 }  
134  
135 if (locale != null) {  
136 this.locale = locale;  
137 }  
138  
139 this.pattern = pattern;  
140 this.locPattern = locPattern;  
141 }  
142  
143 // --------------------------------------------------------- Methods  
144  
145 /\*\*  
146 \* Convert the specified locale-sensitive input object into an output object of the  
147 \* specified type.  
148 \*  
149 \* @param value The input object to be converted  
150 \* @param pattern The pattern is used for the convertion  
151 \* @return The converted value  
152 \*  
153 \* @throws ParseException if conversion cannot be performed  
154 \* successfully  
155 \*/  
156  
157 abstract protected Object parse(Object value, String pattern) throws ParseException;  
158  
159  
160 /\*\*  
161 \* Convert the specified locale-sensitive input object into an output object.  
162 \* The default pattern is used for the conversion.  
163 \*  
164 \* @param value The input object to be converted  
165 \* @return The converted value  
166 \*  
167 \* @throws ConversionException if conversion cannot be performed  
168 \* successfully  
169 \*/  
170 public Object convert(final Object value) {  
171 return convert(value, null);  
172 }  
173  
174 /\*\*  
175 \* Convert the specified locale-sensitive input object into an output object.  
176 \*  
177 \* @param value The input object to be converted  
178 \* @param pattern The pattern is used for the conversion  
179 \* @return The converted value  
180 \*  
181 \* @throws ConversionException if conversion cannot be performed  
182 \* successfully  
183 \*/  
184 public Object convert(final Object value, final String pattern) {  
185 return convert(null, value, pattern);  
186 }  
187  
188 /\*\*  
189 \* Convert the specified locale-sensitive input object into an output object of the  
190 \* specified type. The default pattern is used for the convertion.  
191 \*  
192 \* @param <T> The desired target type of the conversion  
193 \* @param type Data type to which this value should be converted  
194 \* @param value The input object to be converted  
195 \* @return The converted value  
196 \*  
197 \* @throws ConversionException if conversion cannot be performed  
198 \* successfully  
199 \*/  
200 public <T> T convert(final Class<T> type, final Object value) {  
201 return convert(type, value, null);  
202 }  
203  
204 /\*\*  
205 \* Convert the specified locale-sensitive input object into an output object of the  
206 \* specified type.  
207 \*  
208 \* @param <T> The desired target type of the conversion  
209 \* @param type Data is type to which this value should be converted  
210 \* @param value is the input object to be converted  
211 \* @param pattern is the pattern is used for the conversion; if null is  
212 \* passed then the default pattern associated with the converter object  
213 \* will be used.  
214 \* @return The converted value  
215 \*  
216 \* @throws ConversionException if conversion cannot be performed  
217 \* successfully  
218 \*/  
219 public <T> T convert(final Class<T> type, final Object value, final String pattern) {  
220 final Class<T> targetType = ConvertUtils.primitiveToWrapper(type);  
221 if (value == null) {  
222 if (useDefault) {  
223 return getDefaultAs(targetType);  
224 } else {  
225 // symmetric beanutils function allows null  
226 // so do not: throw new ConversionException("No value specified");  
227 log.debug("Null value specified for conversion, returing null");  
228 return null;  
229 }  
230 }  
231  
232 try {  
233 if (pattern != null) {  
234 return checkConversionResult(targetType, parse(value, pattern));  
235 } else {  
236 return checkConversionResult(targetType, parse(value, this.pattern));  
237 }  
238 } catch (final Exception e) {  
239 if (useDefault) {  
240 return getDefaultAs(targetType);  
241 } else {  
242 if (e instanceof ConversionException) {  
243 throw (ConversionException)e;  
244 }  
245 throw new ConversionException(e);  
246 }  
247 }  
248 }  
249  
250 /\*\*  
251 \* Returns the default object specified for this converter cast for the  
252 \* given target type. If the default value is not conform to the given type,  
253 \* an exception is thrown.  
254 \*  
255 \* @param <T> the desired target type  
256 \* @param type the target class of the conversion  
257 \* @return the default value in the given target type  
258 \* @throws ConversionException if the default object is not compatible with  
259 \* the target type  
260 \*/  
261 private <T> T getDefaultAs(final Class<T> type) {  
262 return checkConversionResult(type, defaultValue);  
263 }  
264  
265 /\*\*  
266 \* Checks whether the result of a conversion is conform to the specified  
267 \* target type. If this is the case, the passed in result object is cast to  
268 \* the correct target type. Otherwise, an exception is thrown.  
269 \*  
270 \* @param <T> the desired result type  
271 \* @param type the target class of the conversion  
272 \* @param result the conversion result object  
273 \* @return the result cast to the target class  
274 \* @throws ConversionException if the result object is not compatible with  
275 \* the target type  
276 \*/  
277 private static <T> T checkConversionResult(final Class<T> type, final Object result) {  
278 if (type == null) {  
279 // in this case we cannot do much; the result object is returned  
280 @SuppressWarnings("unchecked")  
281 final  
282 T temp = (T) result;  
283 return temp;  
284 }  
285  
286 if (result == null) {  
287 return null;  
288 }  
289 if (type.isInstance(result)) {  
290 return type.cast(result);  
291 }  
292 throw new ConversionException("Unsupported target type: " + type);  
293 }  
294}