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.lang.reflect.Array;  
021import java.math.BigDecimal;  
022import java.math.BigInteger;  
023import java.util.Collection;  
024import java.util.Locale;  
025import java.util.Map;  
026import java.util.Set;  
027  
028import org.apache.commons.beanutils.BeanUtils;  
029import org.apache.commons.beanutils.locale.converters.BigDecimalLocaleConverter;  
030import org.apache.commons.beanutils.locale.converters.BigIntegerLocaleConverter;  
031import org.apache.commons.beanutils.locale.converters.ByteLocaleConverter;  
032import org.apache.commons.beanutils.locale.converters.DoubleLocaleConverter;  
033import org.apache.commons.beanutils.locale.converters.FloatLocaleConverter;  
034import org.apache.commons.beanutils.locale.converters.IntegerLocaleConverter;  
035import org.apache.commons.beanutils.locale.converters.LongLocaleConverter;  
036import org.apache.commons.beanutils.locale.converters.ShortLocaleConverter;  
037import org.apache.commons.beanutils.locale.converters.SqlDateLocaleConverter;  
038import org.apache.commons.beanutils.locale.converters.SqlTimeLocaleConverter;  
039import org.apache.commons.beanutils.locale.converters.SqlTimestampLocaleConverter;  
040import org.apache.commons.beanutils.locale.converters.StringLocaleConverter;  
041import org.apache.commons.collections.FastHashMap;  
042import org.apache.commons.logging.Log;  
043import org.apache.commons.logging.LogFactory;  
044  
045/\*\*  
046 \* <p>Utility methods for converting locale-sensitive String scalar values to objects of the  
047 \* specified Class, String arrays to arrays of the specified Class and  
048 \* object to locale-sensitive String scalar value.</p>  
049 \*  
050 \* <p>This class provides the implementations used by the static utility methods in  
051 \* {@link LocaleConvertUtils}.</p>  
052 \*  
053 \* <p>The actual {@link LocaleConverter} instance to be used  
054 \* can be registered for each possible destination Class. Unless you override them, standard  
055 \* {@link LocaleConverter} instances are provided for all of the following  
056 \* destination Classes:</p>  
057 \* <ul>  
058 \* <li>java.lang.BigDecimal</li>  
059 \* <li>java.lang.BigInteger</li>  
060 \* <li>byte and java.lang.Byte</li>  
061 \* <li>double and java.lang.Double</li>  
062 \* <li>float and java.lang.Float</li>  
063 \* <li>int and java.lang.Integer</li>  
064 \* <li>long and java.lang.Long</li>  
065 \* <li>short and java.lang.Short</li>  
066 \* <li>java.lang.String</li>  
067 \* <li>java.sql.Date</li>  
068 \* <li>java.sql.Time</li>  
069 \* <li>java.sql.Timestamp</li>  
070 \* </ul>  
071 \*  
072 \* <p>For backwards compatibility, the standard locale converters  
073 \* for primitive types (and the corresponding wrapper classes).  
074 \*  
075 \* If you prefer to have another {@link LocaleConverter}  
076 \* thrown instead, replace the standard {@link LocaleConverter} instances  
077 \* with ones created with the one of the appropriate constructors.  
078 \*  
079 \* It's important that {@link LocaleConverter} should be registered for  
080 \* the specified locale and Class (or primitive type).  
081 \*  
082 \* @since 1.7  
083 \* @version $Id$  
084 \*/  
085public class LocaleConvertUtilsBean {  
086  
087 /\*\*  
088 \* Gets singleton instance.  
089 \* This is the same as the instance used by the default {@link LocaleBeanUtilsBean} singleton.  
090 \* @return the singleton instance  
091 \*/  
092 public static LocaleConvertUtilsBean getInstance() {  
093 return LocaleBeanUtilsBean.getLocaleBeanUtilsInstance().getLocaleConvertUtils();  
094 }  
095  
096 // ----------------------------------------------------- Instance Variables  
097  
098 /\*\* The locale - default for convertion. \*/  
099 private Locale defaultLocale = Locale.getDefault();  
100  
101 /\*\* Indicate whether the pattern is localized or not \*/  
102 private boolean applyLocalized = false;  
103  
104 /\*\* The <code>Log</code> instance for this class. \*/  
105 private final Log log = LogFactory.getLog(LocaleConvertUtils.class);  
106  
107 /\*\* Every entry of the mapConverters is:  
108 \* key = locale  
109 \* value = FastHashMap of converters for the certain locale.  
110 \*/  
111 private final FastHashMap mapConverters = new DelegateFastHashMap(BeanUtils.createCache());  
112  
113 // --------------------------------------------------------- Constructors  
114  
115 /\*\*  
116 \* Makes the state by default (deregisters all converters for all locales)  
117 \* and then registers default locale converters.  
118 \*/  
119 public LocaleConvertUtilsBean() {  
120 mapConverters.setFast(false);  
121 deregister();  
122 mapConverters.setFast(true);  
123 }  
124  
125 // --------------------------------------------------------- Properties  
126  
127 /\*\*  
128 \* getter for defaultLocale.  
129 \* @return the default locale  
130 \*/  
131 public Locale getDefaultLocale() {  
132  
133 return defaultLocale;  
134 }  
135  
136 /\*\*  
137 \* setter for defaultLocale.  
138 \* @param locale the default locale  
139 \*/  
140 public void setDefaultLocale(final Locale locale) {  
141  
142 if (locale == null) {  
143 defaultLocale = Locale.getDefault();  
144 }  
145 else {  
146 defaultLocale = locale;  
147 }  
148 }  
149  
150 /\*\*  
151 \* getter for applyLocalized  
152 \*  
153 \* @return <code>true</code> if pattern is localized,  
154 \* otherwise <code>false</code>  
155 \*/  
156 public boolean getApplyLocalized() {  
157 return applyLocalized;  
158 }  
159  
160 /\*\*  
161 \* setter for applyLocalized  
162 \*  
163 \* @param newApplyLocalized <code>true</code> if pattern is localized,  
164 \* otherwise <code>false</code>  
165 \*/  
166 public void setApplyLocalized(final boolean newApplyLocalized) {  
167 applyLocalized = newApplyLocalized;  
168 }  
169  
170 // --------------------------------------------------------- Methods  
171  
172 /\*\*  
173 \* Convert the specified locale-sensitive value into a String.  
174 \*  
175 \* @param value The Value to be converted  
176 \* @return the converted value  
177 \*  
178 \* @throws org.apache.commons.beanutils.ConversionException if thrown by an  
179 \* underlying Converter  
180 \*/  
181 public String convert(final Object value) {  
182 return convert(value, defaultLocale, null);  
183 }  
184  
185 /\*\*  
186 \* Convert the specified locale-sensitive value into a String  
187 \* using the conversion pattern.  
188 \*  
189 \* @param value The Value to be converted  
190 \* @param pattern The convertion pattern  
191 \* @return the converted value  
192 \*  
193 \* @throws org.apache.commons.beanutils.ConversionException if thrown by an  
194 \* underlying Converter  
195 \*/  
196 public String convert(final Object value, final String pattern) {  
197 return convert(value, defaultLocale, pattern);  
198 }  
199  
200 /\*\*  
201 \* Convert the specified locale-sensitive value into a String  
202 \* using the paticular convertion pattern.  
203 \*  
204 \* @param value The Value to be converted  
205 \* @param locale The locale  
206 \* @param pattern The convertion pattern  
207 \* @return the converted value  
208 \*  
209 \* @throws org.apache.commons.beanutils.ConversionException if thrown by an  
210 \* underlying Converter  
211 \*/  
212 public String convert(final Object value, final Locale locale, final String pattern) {  
213  
214 final LocaleConverter converter = lookup(String.class, locale);  
215  
216 return converter.convert(String.class, value, pattern);  
217 }  
218  
219 /\*\*  
220 \* Convert the specified value to an object of the specified class (if  
221 \* possible). Otherwise, return a String representation of the value.  
222 \*  
223 \* @param value The String scalar value to be converted  
224 \* @param clazz The Data type to which this value should be converted.  
225 \* @return the converted value  
226 \*  
227 \* @throws org.apache.commons.beanutils.ConversionException if thrown by an  
228 \* underlying Converter  
229 \*/  
230 public Object convert(final String value, final Class<?> clazz) {  
231  
232 return convert(value, clazz, defaultLocale, null);  
233 }  
234  
235 /\*\*  
236 \* Convert the specified value to an object of the specified class (if  
237 \* possible) using the convertion pattern. Otherwise, return a String  
238 \* representation of the value.  
239 \*  
240 \* @param value The String scalar value to be converted  
241 \* @param clazz The Data type to which this value should be converted.  
242 \* @param pattern The convertion pattern  
243 \* @return the converted value  
244 \*  
245 \* @throws org.apache.commons.beanutils.ConversionException if thrown by an  
246 \* underlying Converter  
247 \*/  
248 public Object convert(final String value, final Class<?> clazz, final String pattern) {  
249  
250 return convert(value, clazz, defaultLocale, pattern);  
251 }  
252  
253 /\*\*  
254 \* Convert the specified value to an object of the specified class (if  
255 \* possible) using the convertion pattern. Otherwise, return a String  
256 \* representation of the value.  
257 \*  
258 \* @param value The String scalar value to be converted  
259 \* @param clazz The Data type to which this value should be converted.  
260 \* @param locale The locale  
261 \* @param pattern The convertion pattern  
262 \* @return the converted value  
263 \*  
264 \* @throws org.apache.commons.beanutils.ConversionException if thrown by an  
265 \* underlying Converter  
266 \*/  
267 public Object convert(final String value, final Class<?> clazz, final Locale locale, final String pattern) {  
268  
269 if (log.isDebugEnabled()) {  
270 log.debug("Convert string " + value + " to class " +  
271 clazz.getName() + " using " + locale +  
272 " locale and " + pattern + " pattern");  
273 }  
274  
275 Class<?> targetClass = clazz;  
276 LocaleConverter converter = lookup(clazz, locale);  
277  
278 if (converter == null) {  
279 converter = lookup(String.class, locale);  
280 targetClass = String.class;  
281 }  
282 if (log.isTraceEnabled()) {  
283 log.trace(" Using converter " + converter);  
284 }  
285  
286 return (converter.convert(targetClass, value, pattern));  
287 }  
288  
289 /\*\*  
290 \* Convert an array of specified values to an array of objects of the  
291 \* specified class (if possible) using the convertion pattern.  
292 \*  
293 \* @param values Value to be converted (may be null)  
294 \* @param clazz Java array or element class to be converted to  
295 \* @param pattern The convertion pattern  
296 \* @return the converted value  
297 \*  
298 \* @throws org.apache.commons.beanutils.ConversionException if thrown by an  
299 \* underlying Converter  
300 \*/  
301 public Object convert(final String[] values, final Class<?> clazz, final String pattern) {  
302  
303 return convert(values, clazz, getDefaultLocale(), pattern);  
304 }  
305  
306 /\*\*  
307 \* Convert an array of specified values to an array of objects of the  
308 \* specified class (if possible) .  
309 \*  
310 \* @param values Value to be converted (may be null)  
311 \* @param clazz Java array or element class to be converted to  
312 \* @return the converted value  
313 \*  
314 \* @throws org.apache.commons.beanutils.ConversionException if thrown by an  
315 \* underlying Converter  
316 \*/  
317 public Object convert(final String[] values, final Class<?> clazz) {  
318  
319 return convert(values, clazz, getDefaultLocale(), null);  
320 }  
321  
322 /\*\*  
323 \* Convert an array of specified values to an array of objects of the  
324 \* specified class (if possible) using the convertion pattern.  
325 \*  
326 \* @param values Value to be converted (may be null)  
327 \* @param clazz Java array or element class to be converted to  
328 \* @param locale The locale  
329 \* @param pattern The convertion pattern  
330 \* @return the converted value  
331 \*  
332 \* @throws org.apache.commons.beanutils.ConversionException if thrown by an  
333 \* underlying Converter  
334 \*/  
335 public Object convert(final String[] values, final Class<?> clazz, final Locale locale, final String pattern) {  
336  
337 Class<?> type = clazz;  
338 if (clazz.isArray()) {  
339 type = clazz.getComponentType();  
340 }  
341 if (log.isDebugEnabled()) {  
342 log.debug("Convert String[" + values.length + "] to class " +  
343 type.getName() + "[] using " + locale +  
344 " locale and " + pattern + " pattern");  
345 }  
346  
347 final Object array = Array.newInstance(type, values.length);  
348 for (int i = 0; i < values.length; i++) {  
349 Array.set(array, i, convert(values[i], type, locale, pattern));  
350 }  
351  
352 return (array);  
353 }  
354  
355 /\*\*  
356 \* Register a custom {@link LocaleConverter} for the specified destination  
357 \* <code>Class</code>, replacing any previously registered converter.  
358 \*  
359 \* @param converter The LocaleConverter to be registered  
360 \* @param clazz The Destination class for conversions performed by this  
361 \* Converter  
362 \* @param locale The locale  
363 \*/  
364 public void register(final LocaleConverter converter, final Class<?> clazz, final Locale locale) {  
365  
366 lookup(locale).put(clazz, converter);  
367 }  
368  
369 /\*\*  
370 \* Remove any registered {@link LocaleConverter}.  
371 \*/  
372 public void deregister() {  
373  
374 final FastHashMap defaultConverter = lookup(defaultLocale);  
375  
376 mapConverters.setFast(false);  
377  
378 mapConverters.clear();  
379 mapConverters.put(defaultLocale, defaultConverter);  
380  
381 mapConverters.setFast(true);  
382 }  
383  
384  
385 /\*\*  
386 \* Remove any registered {@link LocaleConverter} for the specified locale  
387 \*  
388 \* @param locale The locale  
389 \*/  
390 public void deregister(final Locale locale) {  
391  
392 mapConverters.remove(locale);  
393 }  
394  
395  
396 /\*\*  
397 \* Remove any registered {@link LocaleConverter} for the specified locale and Class.  
398 \*  
399 \* @param clazz Class for which to remove a registered Converter  
400 \* @param locale The locale  
401 \*/  
402 public void deregister(final Class<?> clazz, final Locale locale) {  
403  
404 lookup(locale).remove(clazz);  
405 }  
406  
407 /\*\*  
408 \* Look up and return any registered {@link LocaleConverter} for the specified  
409 \* destination class and locale; if there is no registered Converter, return  
410 \* <code>null</code>.  
411 \*  
412 \* @param clazz Class for which to return a registered Converter  
413 \* @param locale The Locale  
414 \* @return The registered locale Converter, if any  
415 \*/  
416 public LocaleConverter lookup(final Class<?> clazz, final Locale locale) {  
417  
418 final LocaleConverter converter = (LocaleConverter) lookup(locale).get(clazz);  
419  
420 if (log.isTraceEnabled()) {  
421 log.trace("LocaleConverter:" + converter);  
422 }  
423  
424 return converter;  
425 }  
426  
427 /\*\*  
428 \* Look up and return any registered FastHashMap instance for the specified locale;  
429 \* if there is no registered one, return <code>null</code>.  
430 \*  
431 \* @param locale The Locale  
432 \* @return The FastHashMap instance contains the all {@link LocaleConverter} types for  
433 \* the specified locale.  
434 \* @deprecated This method will be modified to return a Map in the next release.  
435 \*/  
436 @Deprecated  
437 protected FastHashMap lookup(final Locale locale) {  
438 FastHashMap localeConverters;  
439  
440 if (locale == null) {  
441 localeConverters = (FastHashMap) mapConverters.get(defaultLocale);  
442 }  
443 else {  
444 localeConverters = (FastHashMap) mapConverters.get(locale);  
445  
446 if (localeConverters == null) {  
447 localeConverters = create(locale);  
448 mapConverters.put(locale, localeConverters);  
449 }  
450 }  
451  
452 return localeConverters;  
453 }  
454  
455 /\*\*  
456 \* Create all {@link LocaleConverter} types for specified locale.  
457 \*  
458 \* @param locale The Locale  
459 \* @return The FastHashMap instance contains the all {@link LocaleConverter} types  
460 \* for the specified locale.  
461 \* @deprecated This method will be modified to return a Map in the next release.  
462 \*/  
463 @Deprecated  
464 protected FastHashMap create(final Locale locale) {  
465  
466 final FastHashMap converter = new DelegateFastHashMap(BeanUtils.createCache());  
467 converter.setFast(false);  
468  
469 converter.put(BigDecimal.class, new BigDecimalLocaleConverter(locale, applyLocalized));  
470 converter.put(BigInteger.class, new BigIntegerLocaleConverter(locale, applyLocalized));  
471  
472 converter.put(Byte.class, new ByteLocaleConverter(locale, applyLocalized));  
473 converter.put(Byte.TYPE, new ByteLocaleConverter(locale, applyLocalized));  
474  
475 converter.put(Double.class, new DoubleLocaleConverter(locale, applyLocalized));  
476 converter.put(Double.TYPE, new DoubleLocaleConverter(locale, applyLocalized));  
477  
478 converter.put(Float.class, new FloatLocaleConverter(locale, applyLocalized));  
479 converter.put(Float.TYPE, new FloatLocaleConverter(locale, applyLocalized));  
480  
481 converter.put(Integer.class, new IntegerLocaleConverter(locale, applyLocalized));  
482 converter.put(Integer.TYPE, new IntegerLocaleConverter(locale, applyLocalized));  
483  
484 converter.put(Long.class, new LongLocaleConverter(locale, applyLocalized));  
485 converter.put(Long.TYPE, new LongLocaleConverter(locale, applyLocalized));  
486  
487 converter.put(Short.class, new ShortLocaleConverter(locale, applyLocalized));  
488 converter.put(Short.TYPE, new ShortLocaleConverter(locale, applyLocalized));  
489  
490 converter.put(String.class, new StringLocaleConverter(locale, applyLocalized));  
491  
492 // conversion format patterns of java.sql.\* types should correspond to default  
493 // behaviour of toString and valueOf methods of these classes  
494 converter.put(java.sql.Date.class, new SqlDateLocaleConverter(locale, "yyyy-MM-dd"));  
495 converter.put(java.sql.Time.class, new SqlTimeLocaleConverter(locale, "HH:mm:ss"));  
496 converter.put( java.sql.Timestamp.class,  
497 new SqlTimestampLocaleConverter(locale, "yyyy-MM-dd HH:mm:ss.S")  
498 );  
499  
500 converter.setFast(true);  
501  
502 return converter;  
503 }  
504  
505 /\*\*  
506 \* FastHashMap implementation that uses WeakReferences to overcome  
507 \* memory leak problems.  
508 \*  
509 \* This is a hack to retain binary compatibility with previous  
510 \* releases (where FastHashMap is exposed in the API), but  
511 \* use WeakHashMap to resolve memory leaks.  
512 \*/  
513 private static class DelegateFastHashMap extends FastHashMap {  
514  
515 private final Map<Object, Object> map;  
516  
517 private DelegateFastHashMap(final Map<Object, Object> map) {  
518 this.map = map;  
519 }  
520 @Override  
521 public void clear() {  
522 map.clear();  
523 }  
524 @Override  
525 public boolean containsKey(final Object key) {  
526 return map.containsKey(key);  
527 }  
528 @Override  
529 public boolean containsValue(final Object value) {  
530 return map.containsValue(value);  
531 }  
532 @Override  
533 public Set<Map.Entry<Object, Object>> entrySet() {  
534 return map.entrySet();  
535 }  
536 @Override  
537 public boolean equals(final Object o) {  
538 return map.equals(o);  
539 }  
540 @Override  
541 public Object get(final Object key) {  
542 return map.get(key);  
543 }  
544 @Override  
545 public int hashCode() {  
546 return map.hashCode();  
547 }  
548 @Override  
549 public boolean isEmpty() {  
550 return map.isEmpty();  
551 }  
552 @Override  
553 public Set<Object> keySet() {  
554 return map.keySet();  
555 }  
556 @Override  
557 public Object put(final Object key, final Object value) {  
558 return map.put(key, value);  
559 }  
560 @SuppressWarnings({ "rawtypes", "unchecked" })  
561 // we operate on very generic types (<Object, Object>), so there is  
562 // no need for doing type checks  
563 @Override  
564 public void putAll(final Map m) {  
565 map.putAll(m);  
566 }  
567 @Override  
568 public Object remove(final Object key) {  
569 return map.remove(key);  
570 }  
571 @Override  
572 public int size() {  
573 return map.size();  
574 }  
575 @Override  
576 public Collection<Object> values() {  
577 return map.values();  
578 }  
579 @Override  
580 public boolean getFast() {  
581 return BeanUtils.getCacheFast(map);  
582 }  
583 @Override  
584 public void setFast(final boolean fast) {  
585 BeanUtils.setCacheFast(map, fast);  
586 }  
587 }  
588}