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;  
020  
021  
022import java.io.File;  
023import java.lang.reflect.Array;  
024import java.math.BigDecimal;  
025import java.math.BigInteger;  
026import java.net.URL;  
027import java.sql.Timestamp;  
028import java.util.Calendar;  
029import java.util.Collection;  
030  
031import org.apache.commons.beanutils.converters.ArrayConverter;  
032import org.apache.commons.beanutils.converters.BigDecimalConverter;  
033import org.apache.commons.beanutils.converters.BigIntegerConverter;  
034import org.apache.commons.beanutils.converters.BooleanConverter;  
035import org.apache.commons.beanutils.converters.ByteConverter;  
036import org.apache.commons.beanutils.converters.CalendarConverter;  
037import org.apache.commons.beanutils.converters.CharacterConverter;  
038import org.apache.commons.beanutils.converters.ClassConverter;  
039import org.apache.commons.beanutils.converters.ConverterFacade;  
040import org.apache.commons.beanutils.converters.DateConverter;  
041import org.apache.commons.beanutils.converters.DoubleConverter;  
042import org.apache.commons.beanutils.converters.FileConverter;  
043import org.apache.commons.beanutils.converters.FloatConverter;  
044import org.apache.commons.beanutils.converters.IntegerConverter;  
045import org.apache.commons.beanutils.converters.LongConverter;  
046import org.apache.commons.beanutils.converters.ShortConverter;  
047import org.apache.commons.beanutils.converters.SqlDateConverter;  
048import org.apache.commons.beanutils.converters.SqlTimeConverter;  
049import org.apache.commons.beanutils.converters.SqlTimestampConverter;  
050import org.apache.commons.beanutils.converters.StringConverter;  
051import org.apache.commons.beanutils.converters.URLConverter;  
052import org.apache.commons.logging.Log;  
053import org.apache.commons.logging.LogFactory;  
054  
055  
056/\*\*  
057 \* <p>Utility methods for converting String scalar values to objects of the  
058 \* specified Class, String arrays to arrays of the specified Class. The  
059 \* actual {@link Converter} instance to be used can be registered for each  
060 \* possible destination Class. Unless you override them, standard  
061 \* {@link Converter} instances are provided for all of the following  
062 \* destination Classes:</p>  
063 \* <ul>  
064 \* <li>java.lang.BigDecimal (no default value)</li>  
065 \* <li>java.lang.BigInteger (no default value)</li>  
066 \* <li>boolean and java.lang.Boolean (default to false)</li>  
067 \* <li>byte and java.lang.Byte (default to zero)</li>  
068 \* <li>char and java.lang.Character (default to a space)</li>  
069 \* <li>java.lang.Class (no default value)</li>  
070 \* <li>double and java.lang.Double (default to zero)</li>  
071 \* <li>float and java.lang.Float (default to zero)</li>  
072 \* <li>int and java.lang.Integer (default to zero)</li>  
073 \* <li>long and java.lang.Long (default to zero)</li>  
074 \* <li>short and java.lang.Short (default to zero)</li>  
075 \* <li>java.lang.String (default to null)</li>  
076 \* <li>java.io.File (no default value)</li>  
077 \* <li>java.net.URL (no default value)</li>  
078 \* <li>java.sql.Date (no default value)</li>  
079 \* <li>java.sql.Time (no default value)</li>  
080 \* <li>java.sql.Timestamp (no default value)</li>  
081 \* </ul>  
082 \*  
083 \* <p>For backwards compatibility, the standard Converters for primitive  
084 \* types (and the corresponding wrapper classes) return a defined  
085 \* default value when a conversion error occurs. If you prefer to have a  
086 \* {@link ConversionException} thrown instead, replace the standard Converter  
087 \* instances with instances created with the zero-arguments constructor. For  
088 \* example, to cause the Converters for integers to throw an exception on  
089 \* conversion errors, you could do this:</p>  
090 \* <pre>  
091 \* // No-args constructor gets the version that throws exceptions  
092 \* Converter myConverter =  
093 \* new org.apache.commons.beanutils.converter.IntegerConverter();  
094 \* ConvertUtils.register(myConverter, Integer.TYPE); // Native type  
095 \* ConvertUtils.register(myConverter, Integer.class); // Wrapper class  
096 \* </pre>  
097 \*  
098 \* <p>  
099 \* Converters generally treat null input as if it were invalid  
100 \* input, ie they return their default value if one was specified when the  
101 \* converter was constructed, and throw an exception otherwise. If you prefer  
102 \* nulls to be preserved for converters that are converting to objects (not  
103 \* primitives) then register a converter as above, passing a default value of  
104 \* null to the converter constructor (and of course registering that converter  
105 \* only for the .class target).  
106 \* </p>  
107 \*  
108 \* <p>  
109 \* When a converter is listed above as having no default value, then that  
110 \* converter will throw an exception when passed null or an invalid value  
111 \* as its input. In particular, by default the BigInteger and BigDecimal  
112 \* converters have no default (and are therefore somewhat inconsistent  
113 \* with the other numerical converters which all have zero as their default).  
114 \* </p>  
115 \*  
116 \* <p>  
117 \* Converters that generate <i>arrays</i> of each of the primitive types are  
118 \* also automatically configured (including String[]). When passed null  
119 \* or invalid input, these return an empty array (not null). See class  
120 \* AbstractArrayConverter for the supported input formats for these converters.  
121 \* </p>  
122 \*  
123 \* @version $Id$  
124 \* @since 1.7  
125 \*/  
126  
127public class ConvertUtilsBean {  
128  
129 private static final Integer ZERO = new Integer(0);  
130 private static final Character SPACE = new Character(' ');  
131  
132 // ------------------------------------------------------- Class Methods  
133 /\*\*  
134 \* Get singleton instance  
135 \* @return The singleton instance  
136 \*/  
137 protected static ConvertUtilsBean getInstance() {  
138 return BeanUtilsBean.getInstance().getConvertUtils();  
139 }  
140  
141 // ------------------------------------------------------- Variables  
142  
143  
144 /\*\*  
145 \* The set of {@link Converter}s that can be used to convert Strings  
146 \* into objects of a specified Class, keyed by the destination Class.  
147 \*/  
148 private final WeakFastHashMap<Class<?>, Converter> converters =  
149 new WeakFastHashMap<Class<?>, Converter>();  
150  
151 /\*\*  
152 \* The <code>Log</code> instance for this class.  
153 \*/  
154 private final Log log = LogFactory.getLog(ConvertUtils.class);  
155  
156 // ------------------------------------------------------- Constructors  
157  
158 /\*\* Construct a bean with standard converters registered \*/  
159 public ConvertUtilsBean() {  
160 converters.setFast(false);  
161 deregister();  
162 converters.setFast(true);  
163 }  
164  
165 // --------------------------------------------------------- Public Methods  
166  
167 /\*\*  
168 \* The default value for Boolean conversions.  
169 \* @deprecated Register replacement converters for Boolean.TYPE and  
170 \* Boolean.class instead  
171 \*/  
172 @Deprecated  
173 private Boolean defaultBoolean = Boolean.FALSE;  
174  
175 /\*\*  
176 \* Gets the default value for Boolean conversions.  
177 \* @return The default Boolean value  
178 \* @deprecated Register replacement converters for Boolean.TYPE and  
179 \* Boolean.class instead  
180 \*/  
181 @Deprecated  
182 public boolean getDefaultBoolean() {  
183 return (defaultBoolean.booleanValue());  
184 }  
185  
186 /\*\*  
187 \* Sets the default value for Boolean conversions.  
188 \* @param newDefaultBoolean The default Boolean value  
189 \* @deprecated Register replacement converters for Boolean.TYPE and  
190 \* Boolean.class instead  
191 \*/  
192 @Deprecated  
193 public void setDefaultBoolean(final boolean newDefaultBoolean) {  
194 defaultBoolean = (newDefaultBoolean ? Boolean.TRUE : Boolean.FALSE);  
195 register(new BooleanConverter(defaultBoolean), Boolean.TYPE);  
196 register(new BooleanConverter(defaultBoolean), Boolean.class);  
197 }  
198  
199  
200 /\*\*  
201 \* The default value for Byte conversions.  
202 \* @deprecated Register replacement converters for Byte.TYPE and  
203 \* Byte.class instead  
204 \*/  
205 @Deprecated  
206 private Byte defaultByte = new Byte((byte) 0);  
207  
208 /\*\*  
209 \* Gets the default value for Byte conversions.  
210 \* @return The default Byte value  
211 \* @deprecated Register replacement converters for Byte.TYPE and  
212 \* Byte.class instead  
213 \*/  
214 @Deprecated  
215 public byte getDefaultByte() {  
216 return (defaultByte.byteValue());  
217 }  
218  
219 /\*\*  
220 \* Sets the default value for Byte conversions.  
221 \* @param newDefaultByte The default Byte value  
222 \* @deprecated Register replacement converters for Byte.TYPE and  
223 \* Byte.class instead  
224 \*/  
225 @Deprecated  
226 public void setDefaultByte(final byte newDefaultByte) {  
227 defaultByte = new Byte(newDefaultByte);  
228 register(new ByteConverter(defaultByte), Byte.TYPE);  
229 register(new ByteConverter(defaultByte), Byte.class);  
230 }  
231  
232  
233 /\*\*  
234 \* The default value for Character conversions.  
235 \* @deprecated Register replacement converters for Character.TYPE and  
236 \* Character.class instead  
237 \*/  
238 @Deprecated  
239 private Character defaultCharacter = new Character(' ');  
240  
241 /\*\*  
242 \* Gets the default value for Character conversions.  
243 \* @return The default Character value  
244 \* @deprecated Register replacement converters for Character.TYPE and  
245 \* Character.class instead  
246 \*/  
247 @Deprecated  
248 public char getDefaultCharacter() {  
249 return (defaultCharacter.charValue());  
250 }  
251  
252 /\*\*  
253 \* Sets the default value for Character conversions.  
254 \* @param newDefaultCharacter The default Character value  
255 \* @deprecated Register replacement converters for Character.TYPE and  
256 \* Character.class instead  
257 \*/  
258 @Deprecated  
259 public void setDefaultCharacter(final char newDefaultCharacter) {  
260 defaultCharacter = new Character(newDefaultCharacter);  
261 register(new CharacterConverter(defaultCharacter),  
262 Character.TYPE);  
263 register(new CharacterConverter(defaultCharacter),  
264 Character.class);  
265 }  
266  
267  
268 /\*\*  
269 \* The default value for Double conversions.  
270 \* @deprecated Register replacement converters for Double.TYPE and  
271 \* Double.class instead  
272 \*/  
273 @Deprecated  
274 private Double defaultDouble = new Double(0.0);  
275  
276 /\*\*  
277 \* Gets the default value for Double conversions.  
278 \* @return The default Double value  
279 \* @deprecated Register replacement converters for Double.TYPE and  
280 \* Double.class instead  
281 \*/  
282 @Deprecated  
283 public double getDefaultDouble() {  
284 return (defaultDouble.doubleValue());  
285 }  
286  
287 /\*\*  
288 \* Sets the default value for Double conversions.  
289 \* @param newDefaultDouble The default Double value  
290 \* @deprecated Register replacement converters for Double.TYPE and  
291 \* Double.class instead  
292 \*/  
293 @Deprecated  
294 public void setDefaultDouble(final double newDefaultDouble) {  
295 defaultDouble = new Double(newDefaultDouble);  
296 register(new DoubleConverter(defaultDouble), Double.TYPE);  
297 register(new DoubleConverter(defaultDouble), Double.class);  
298 }  
299  
300  
301 /\*\*  
302 \* The default value for Float conversions.  
303 \* @deprecated Register replacement converters for Float.TYPE and  
304 \* Float.class instead  
305 \*/  
306 @Deprecated  
307 private Float defaultFloat = new Float((float) 0.0);  
308  
309 /\*\*  
310 \* Gets the default value for Float conversions.  
311 \* @return The default Float value  
312 \* @deprecated Register replacement converters for Float.TYPE and  
313 \* Float.class instead  
314 \*/  
315 @Deprecated  
316 public float getDefaultFloat() {  
317 return (defaultFloat.floatValue());  
318 }  
319  
320 /\*\*  
321 \* Sets the default value for Float conversions.  
322 \* @param newDefaultFloat The default Float value  
323 \* @deprecated Register replacement converters for Float.TYPE and  
324 \* Float.class instead  
325 \*/  
326 @Deprecated  
327 public void setDefaultFloat(final float newDefaultFloat) {  
328 defaultFloat = new Float(newDefaultFloat);  
329 register(new FloatConverter(defaultFloat), Float.TYPE);  
330 register(new FloatConverter(defaultFloat), Float.class);  
331 }  
332  
333  
334 /\*\*  
335 \* The default value for Integer conversions.  
336 \* @deprecated Register replacement converters for Integer.TYPE and  
337 \* Integer.class instead  
338 \*/  
339 @Deprecated  
340 private Integer defaultInteger = new Integer(0);  
341  
342 /\*\*  
343 \* Gets the default value for Integer conversions.  
344 \* @return The default Integer value  
345 \* @deprecated Register replacement converters for Integer.TYPE and  
346 \* Integer.class instead  
347 \*/  
348 @Deprecated  
349 public int getDefaultInteger() {  
350 return (defaultInteger.intValue());  
351 }  
352  
353 /\*\*  
354 \* Sets the default value for Integer conversions.  
355 \* @param newDefaultInteger The default Integer value  
356 \* @deprecated Register replacement converters for Integer.TYPE and  
357 \* Integer.class instead  
358 \*/  
359 @Deprecated  
360 public void setDefaultInteger(final int newDefaultInteger) {  
361 defaultInteger = new Integer(newDefaultInteger);  
362 register(new IntegerConverter(defaultInteger), Integer.TYPE);  
363 register(new IntegerConverter(defaultInteger), Integer.class);  
364 }  
365  
366  
367 /\*\*  
368 \* The default value for Long conversions.  
369 \* @deprecated Register replacement converters for Long.TYPE and  
370 \* Long.class instead  
371 \*/  
372 @Deprecated  
373 private Long defaultLong = new Long(0);  
374  
375 /\*\*  
376 \* Gets the default value for Long conversions.  
377 \* @return The default Long value  
378 \* @deprecated Register replacement converters for Long.TYPE and  
379 \* Long.class instead  
380 \*/  
381 @Deprecated  
382 public long getDefaultLong() {  
383 return (defaultLong.longValue());  
384 }  
385  
386 /\*\*  
387 \* Sets the default value for Long conversions.  
388 \* @param newDefaultLong The default Long value  
389 \* @deprecated Register replacement converters for Long.TYPE and  
390 \* Long.class instead  
391 \*/  
392 @Deprecated  
393 public void setDefaultLong(final long newDefaultLong) {  
394 defaultLong = new Long(newDefaultLong);  
395 register(new LongConverter(defaultLong), Long.TYPE);  
396 register(new LongConverter(defaultLong), Long.class);  
397 }  
398  
399  
400 /\*\*  
401 \* The default value for Short conversions.  
402 \* @deprecated Register replacement converters for Short.TYPE and  
403 \* Short.class instead  
404 \*/  
405 @Deprecated  
406 private static Short defaultShort = new Short((short) 0);  
407  
408 /\*\*  
409 \* Gets the default value for Short conversions.  
410 \* @return The default Short value  
411 \* @deprecated Register replacement converters for Short.TYPE and  
412 \* Short.class instead  
413 \*/  
414 @Deprecated  
415 public short getDefaultShort() {  
416 return (defaultShort.shortValue());  
417 }  
418  
419 /\*\*  
420 \* Sets the default value for Short conversions.  
421 \* @param newDefaultShort The default Short value  
422 \* @deprecated Register replacement converters for Short.TYPE and  
423 \* Short.class instead  
424 \*/  
425 @Deprecated  
426 public void setDefaultShort(final short newDefaultShort) {  
427 defaultShort = new Short(newDefaultShort);  
428 register(new ShortConverter(defaultShort), Short.TYPE);  
429 register(new ShortConverter(defaultShort), Short.class);  
430 }  
431  
432  
433  
434 /\*\*  
435 \* Convert the specified value into a String. If the specified value  
436 \* is an array, the first element (converted to a String) will be  
437 \* returned. The registered {@link Converter} for the  
438 \* <code>java.lang.String</code> class will be used, which allows  
439 \* applications to customize Object->String conversions (the default  
440 \* implementation simply uses toString()).  
441 \*  
442 \* @param value Value to be converted (may be null)  
443 \* @return The converted String value or null if value is null  
444 \*/  
445 public String convert(Object value) {  
446  
447 if (value == null) {  
448 return null;  
449 } else if (value.getClass().isArray()) {  
450 if (Array.getLength(value) < 1) {  
451 return (null);  
452 }  
453 value = Array.get(value, 0);  
454 if (value == null) {  
455 return null;  
456 } else {  
457 final Converter converter = lookup(String.class);  
458 return (converter.convert(String.class, value));  
459 }  
460 } else {  
461 final Converter converter = lookup(String.class);  
462 return (converter.convert(String.class, value));  
463 }  
464  
465 }  
466  
467  
468 /\*\*  
469 \* Convert the specified value to an object of the specified class (if  
470 \* possible). Otherwise, return a String representation of the value.  
471 \*  
472 \* @param value Value to be converted (may be null)  
473 \* @param clazz Java class to be converted to (must not be null)  
474 \* @return The converted value  
475 \*  
476 \* @throws ConversionException if thrown by an underlying Converter  
477 \*/  
478 public Object convert(final String value, final Class<?> clazz) {  
479  
480 if (log.isDebugEnabled()) {  
481 log.debug("Convert string '" + value + "' to class '" +  
482 clazz.getName() + "'");  
483 }  
484 Converter converter = lookup(clazz);  
485 if (converter == null) {  
486 converter = lookup(String.class);  
487 }  
488 if (log.isTraceEnabled()) {  
489 log.trace(" Using converter " + converter);  
490 }  
491 return (converter.convert(clazz, value));  
492  
493 }  
494  
495  
496 /\*\*  
497 \* Convert an array of specified values to an array of objects of the  
498 \* specified class (if possible). If the specified Java class is itself  
499 \* an array class, this class will be the type of the returned value.  
500 \* Otherwise, an array will be constructed whose component type is the  
501 \* specified class.  
502 \*  
503 \* @param values Array of values to be converted  
504 \* @param clazz Java array or element class to be converted to (must not be null)  
505 \* @return The converted value  
506 \*  
507 \* @throws ConversionException if thrown by an underlying Converter  
508 \*/  
509 public Object convert(final String[] values, final Class<?> clazz) {  
510  
511 Class<?> type = clazz;  
512 if (clazz.isArray()) {  
513 type = clazz.getComponentType();  
514 }  
515 if (log.isDebugEnabled()) {  
516 log.debug("Convert String[" + values.length + "] to class '" +  
517 type.getName() + "[]'");  
518 }  
519 Converter converter = lookup(type);  
520 if (converter == null) {  
521 converter = lookup(String.class);  
522 }  
523 if (log.isTraceEnabled()) {  
524 log.trace(" Using converter " + converter);  
525 }  
526 final Object array = Array.newInstance(type, values.length);  
527 for (int i = 0; i < values.length; i++) {  
528 Array.set(array, i, converter.convert(type, values[i]));  
529 }  
530 return (array);  
531  
532 }  
533  
534  
535 /\*\*  
536 \* Convert the value to an object of the specified class (if  
537 \* possible). If no converter for the desired target type is registered,  
538 \* the passed in object is returned unchanged.  
539 \*  
540 \* @param value Value to be converted (may be null)  
541 \* @param targetType Class of the value to be converted to (must not be null)  
542 \* @return The converted value  
543 \*  
544 \* @throws ConversionException if thrown by an underlying Converter  
545 \*/  
546 public Object convert(final Object value, final Class<?> targetType) {  
547  
548 final Class<?> sourceType = value == null ? null : value.getClass();  
549  
550 if (log.isDebugEnabled()) {  
551 if (value == null) {  
552 log.debug("Convert null value to type '" +  
553 targetType.getName() + "'");  
554 } else {  
555 log.debug("Convert type '" + sourceType.getName() + "' value '" + value +  
556 "' to type '" + targetType.getName() + "'");  
557 }  
558 }  
559  
560 Object converted = value;  
561 Converter converter = lookup(sourceType, targetType);  
562 if (converter != null) {  
563 if (log.isTraceEnabled()) {  
564 log.trace(" Using converter " + converter);  
565 }  
566 converted = converter.convert(targetType, value);  
567 }  
568 if (String.class.equals(targetType) && converted != null &&  
569 !(converted instanceof String)) {  
570  
571 // NOTE: For backwards compatibility, if the Converter  
572 // doesn't handle conversion-->String then  
573 // use the registered String Converter  
574 converter = lookup(String.class);  
575 if (converter != null) {  
576 if (log.isTraceEnabled()) {  
577 log.trace(" Using converter " + converter);  
578 }  
579 converted = converter.convert(String.class, converted);  
580 }  
581  
582 // If the object still isn't a String, use toString() method  
583 if (converted != null && !(converted instanceof String)) {  
584 converted = converted.toString();  
585 }  
586  
587 }  
588 return converted;  
589  
590 }  
591  
592 /\*\*  
593 \* Remove all registered {@link Converter}s, and re-establish the  
594 \* standard Converters.  
595 \*/  
596 public void deregister() {  
597  
598 converters.clear();  
599  
600 registerPrimitives(false);  
601 registerStandard(false, false);  
602 registerOther(true);  
603 registerArrays(false, 0);  
604 register(BigDecimal.class, new BigDecimalConverter());  
605 register(BigInteger.class, new BigIntegerConverter());  
606 }  
607  
608 /\*\*  
609 \* Register the provided converters with the specified defaults.  
610 \*  
611 \* @param throwException <code>true</code> if the converters should  
612 \* throw an exception when a conversion error occurs, otherwise  
613 \* <code>false</code> if a default value should be used.  
614 \* @param defaultNull <code>true</code>if the <i>standard</i> converters  
615 \* (see {@link ConvertUtilsBean#registerStandard(boolean, boolean)})  
616 \* should use a default value of <code>null</code>, otherwise <code>false</code>.  
617 \* N.B. This values is ignored if <code>throwException</code> is <code>true</code>  
618 \* @param defaultArraySize The size of the default array value for array converters  
619 \* (N.B. This values is ignored if <code>throwException</code> is <code>true</code>).  
620 \* Specifying a value less than zero causes a <code>null</code> value to be used for  
621 \* the default.  
622 \*/  
623 public void register(final boolean throwException, final boolean defaultNull, final int defaultArraySize) {  
624 registerPrimitives(throwException);  
625 registerStandard(throwException, defaultNull);  
626 registerOther(throwException);  
627 registerArrays(throwException, defaultArraySize);  
628 }  
629  
630 /\*\*  
631 \* Register the converters for primitive types.  
632 \* </p>  
633 \* This method registers the following converters:  
634 \* <ul>  
635 \* <li><code>Boolean.TYPE</code> - {@link BooleanConverter}</li>  
636 \* <li><code>Byte.TYPE</code> - {@link ByteConverter}</li>  
637 \* <li><code>Character.TYPE</code> - {@link CharacterConverter}</li>  
638 \* <li><code>Double.TYPE</code> - {@link DoubleConverter}</li>  
639 \* <li><code>Float.TYPE</code> - {@link FloatConverter}</li>  
640 \* <li><code>Integer.TYPE</code> - {@link IntegerConverter}</li>  
641 \* <li><code>Long.TYPE</code> - {@link LongConverter}</li>  
642 \* <li><code>Short.TYPE</code> - {@link ShortConverter}</li>  
643 \* </ul>  
644 \* @param throwException <code>true</code> if the converters should  
645 \* throw an exception when a conversion error occurs, otherwise <code>  
646 \* <code>false</code> if a default value should be used.  
647 \*/  
648 private void registerPrimitives(final boolean throwException) {  
649 register(Boolean.TYPE, throwException ? new BooleanConverter() : new BooleanConverter(Boolean.FALSE));  
650 register(Byte.TYPE, throwException ? new ByteConverter() : new ByteConverter(ZERO));  
651 register(Character.TYPE, throwException ? new CharacterConverter() : new CharacterConverter(SPACE));  
652 register(Double.TYPE, throwException ? new DoubleConverter() : new DoubleConverter(ZERO));  
653 register(Float.TYPE, throwException ? new FloatConverter() : new FloatConverter(ZERO));  
654 register(Integer.TYPE, throwException ? new IntegerConverter() : new IntegerConverter(ZERO));  
655 register(Long.TYPE, throwException ? new LongConverter() : new LongConverter(ZERO));  
656 register(Short.TYPE, throwException ? new ShortConverter() : new ShortConverter(ZERO));  
657 }  
658  
659 /\*\*  
660 \* Register the converters for standard types.  
661 \* </p>  
662 \* This method registers the following converters:  
663 \* <ul>  
664 \* <li><code>BigDecimal.class</code> - {@link BigDecimalConverter}</li>  
665 \* <li><code>BigInteger.class</code> - {@link BigIntegerConverter}</li>  
666 \* <li><code>Boolean.class</code> - {@link BooleanConverter}</li>  
667 \* <li><code>Byte.class</code> - {@link ByteConverter}</li>  
668 \* <li><code>Character.class</code> - {@link CharacterConverter}</li>  
669 \* <li><code>Double.class</code> - {@link DoubleConverter}</li>  
670 \* <li><code>Float.class</code> - {@link FloatConverter}</li>  
671 \* <li><code>Integer.class</code> - {@link IntegerConverter}</li>  
672 \* <li><code>Long.class</code> - {@link LongConverter}</li>  
673 \* <li><code>Short.class</code> - {@link ShortConverter}</li>  
674 \* <li><code>String.class</code> - {@link StringConverter}</li>  
675 \* </ul>  
676 \* @param throwException <code>true</code> if the converters should  
677 \* throw an exception when a conversion error occurs, otherwise <code>  
678 \* <code>false</code> if a default value should be used.  
679 \* @param defaultNull <code>true</code>if the <i>standard</i> converters  
680 \* (see {@link ConvertUtilsBean#registerStandard(boolean, boolean)})  
681 \* should use a default value of <code>null</code>, otherwise <code>false</code>.  
682 \* N.B. This values is ignored if <code>throwException</code> is <code>true</code>  
683 \*/  
684 private void registerStandard(final boolean throwException, final boolean defaultNull) {  
685  
686 final Number defaultNumber = defaultNull ? null : ZERO;  
687 final BigDecimal bigDecDeflt = defaultNull ? null : new BigDecimal("0.0");  
688 final BigInteger bigIntDeflt = defaultNull ? null : new BigInteger("0");  
689 final Boolean booleanDefault = defaultNull ? null : Boolean.FALSE;  
690 final Character charDefault = defaultNull ? null : SPACE;  
691 final String stringDefault = defaultNull ? null : "";  
692  
693 register(BigDecimal.class, throwException ? new BigDecimalConverter() : new BigDecimalConverter(bigDecDeflt));  
694 register(BigInteger.class, throwException ? new BigIntegerConverter() : new BigIntegerConverter(bigIntDeflt));  
695 register(Boolean.class, throwException ? new BooleanConverter() : new BooleanConverter(booleanDefault));  
696 register(Byte.class, throwException ? new ByteConverter() : new ByteConverter(defaultNumber));  
697 register(Character.class, throwException ? new CharacterConverter() : new CharacterConverter(charDefault));  
698 register(Double.class, throwException ? new DoubleConverter() : new DoubleConverter(defaultNumber));  
699 register(Float.class, throwException ? new FloatConverter() : new FloatConverter(defaultNumber));  
700 register(Integer.class, throwException ? new IntegerConverter() : new IntegerConverter(defaultNumber));  
701 register(Long.class, throwException ? new LongConverter() : new LongConverter(defaultNumber));  
702 register(Short.class, throwException ? new ShortConverter() : new ShortConverter(defaultNumber));  
703 register(String.class, throwException ? new StringConverter() : new StringConverter(stringDefault));  
704  
705 }  
706  
707 /\*\*  
708 \* Register the converters for other types.  
709 \* </p>  
710 \* This method registers the following converters:  
711 \* <ul>  
712 \* <li><code>Class.class</code> - {@link ClassConverter}</li>  
713 \* <li><code>java.util.Date.class</code> - {@link DateConverter}</li>  
714 \* <li><code>java.util.Calendar.class</code> - {@link CalendarConverter}</li>  
715 \* <li><code>File.class</code> - {@link FileConverter}</li>  
716 \* <li><code>java.sql.Date.class</code> - {@link SqlDateConverter}</li>  
717 \* <li><code>java.sql.Time.class</code> - {@link SqlTimeConverter}</li>  
718 \* <li><code>java.sql.Timestamp.class</code> - {@link SqlTimestampConverter}</li>  
719 \* <li><code>URL.class</code> - {@link URLConverter}</li>  
720 \* </ul>  
721 \* @param throwException <code>true</code> if the converters should  
722 \* throw an exception when a conversion error occurs, otherwise <code>  
723 \* <code>false</code> if a default value should be used.  
724 \*/  
725 private void registerOther(final boolean throwException) {  
726 register(Class.class, throwException ? new ClassConverter() : new ClassConverter(null));  
727 register(java.util.Date.class, throwException ? new DateConverter() : new DateConverter(null));  
728 register(Calendar.class, throwException ? new CalendarConverter() : new CalendarConverter(null));  
729 register(File.class, throwException ? new FileConverter() : new FileConverter(null));  
730 register(java.sql.Date.class, throwException ? new SqlDateConverter() : new SqlDateConverter(null));  
731 register(java.sql.Time.class, throwException ? new SqlTimeConverter() : new SqlTimeConverter(null));  
732 register(Timestamp.class, throwException ? new SqlTimestampConverter() : new SqlTimestampConverter(null));  
733 register(URL.class, throwException ? new URLConverter() : new URLConverter(null));  
734 }  
735  
736 /\*\*  
737 \* Register array converters.  
738 \*  
739 \* @param throwException <code>true</code> if the converters should  
740 \* throw an exception when a conversion error occurs, otherwise <code>  
741 \* <code>false</code> if a default value should be used.  
742 \* @param defaultArraySize The size of the default array value for array converters  
743 \* (N.B. This values is ignored if <code>throwException</code> is <code>true</code>).  
744 \* Specifying a value less than zero causes a <code>null<code> value to be used for  
745 \* the default.  
746 \*/  
747 private void registerArrays(final boolean throwException, final int defaultArraySize) {  
748  
749 // Primitives  
750 registerArrayConverter(Boolean.TYPE, new BooleanConverter(), throwException, defaultArraySize);  
751 registerArrayConverter(Byte.TYPE, new ByteConverter(), throwException, defaultArraySize);  
752 registerArrayConverter(Character.TYPE, new CharacterConverter(), throwException, defaultArraySize);  
753 registerArrayConverter(Double.TYPE, new DoubleConverter(), throwException, defaultArraySize);  
754 registerArrayConverter(Float.TYPE, new FloatConverter(), throwException, defaultArraySize);  
755 registerArrayConverter(Integer.TYPE, new IntegerConverter(), throwException, defaultArraySize);  
756 registerArrayConverter(Long.TYPE, new LongConverter(), throwException, defaultArraySize);  
757 registerArrayConverter(Short.TYPE, new ShortConverter(), throwException, defaultArraySize);  
758  
759 // Standard  
760 registerArrayConverter(BigDecimal.class, new BigDecimalConverter(), throwException, defaultArraySize);  
761 registerArrayConverter(BigInteger.class, new BigIntegerConverter(), throwException, defaultArraySize);  
762 registerArrayConverter(Boolean.class, new BooleanConverter(), throwException, defaultArraySize);  
763 registerArrayConverter(Byte.class, new ByteConverter(), throwException, defaultArraySize);  
764 registerArrayConverter(Character.class, new CharacterConverter(), throwException, defaultArraySize);  
765 registerArrayConverter(Double.class, new DoubleConverter(), throwException, defaultArraySize);  
766 registerArrayConverter(Float.class, new FloatConverter(), throwException, defaultArraySize);  
767 registerArrayConverter(Integer.class, new IntegerConverter(), throwException, defaultArraySize);  
768 registerArrayConverter(Long.class, new LongConverter(), throwException, defaultArraySize);  
769 registerArrayConverter(Short.class, new ShortConverter(), throwException, defaultArraySize);  
770 registerArrayConverter(String.class, new StringConverter(), throwException, defaultArraySize);  
771  
772 // Other  
773 registerArrayConverter(Class.class, new ClassConverter(), throwException, defaultArraySize);  
774 registerArrayConverter(java.util.Date.class, new DateConverter(), throwException, defaultArraySize);  
775 registerArrayConverter(Calendar.class, new DateConverter(), throwException, defaultArraySize);  
776 registerArrayConverter(File.class, new FileConverter(), throwException, defaultArraySize);  
777 registerArrayConverter(java.sql.Date.class, new SqlDateConverter(), throwException, defaultArraySize);  
778 registerArrayConverter(java.sql.Time.class, new SqlTimeConverter(), throwException, defaultArraySize);  
779 registerArrayConverter(Timestamp.class, new SqlTimestampConverter(), throwException, defaultArraySize);  
780 registerArrayConverter(URL.class, new URLConverter(), throwException, defaultArraySize);  
781  
782 }  
783  
784 /\*\*  
785 \* Register a new ArrayConverter with the specified element delegate converter  
786 \* that returns a default array of the specified size in the event of conversion errors.  
787 \*  
788 \* @param componentType The component type of the array  
789 \* @param componentConverter The converter to delegate to for the array elements  
790 \* @param throwException Whether a conversion exception should be thrown or a default  
791 \* value used in the event of a conversion error  
792 \* @param defaultArraySize The size of the default array  
793 \*/  
794 private void registerArrayConverter(final Class<?> componentType, final Converter componentConverter,  
795 final boolean throwException, final int defaultArraySize) {  
796 final Class<?> arrayType = Array.newInstance(componentType, 0).getClass();  
797 Converter arrayConverter = null;  
798 if (throwException) {  
799 arrayConverter = new ArrayConverter(arrayType, componentConverter);  
800 } else {  
801 arrayConverter = new ArrayConverter(arrayType, componentConverter, defaultArraySize);  
802 }  
803 register(arrayType, arrayConverter);  
804 }  
805  
806 /\*\* strictly for convenience since it has same parameter order as Map.put \*/  
807 private void register(final Class<?> clazz, final Converter converter) {  
808 register(new ConverterFacade(converter), clazz);  
809 }  
810  
811 /\*\*  
812 \* Remove any registered {@link Converter} for the specified destination  
813 \* <code>Class</code>.  
814 \*  
815 \* @param clazz Class for which to remove a registered Converter  
816 \*/  
817 public void deregister(final Class<?> clazz) {  
818  
819 converters.remove(clazz);  
820  
821 }  
822  
823  
824 /\*\*  
825 \* Look up and return any registered {@link Converter} for the specified  
826 \* destination class; if there is no registered Converter, return  
827 \* <code>null</code>.  
828 \*  
829 \* @param clazz Class for which to return a registered Converter  
830 \* @return The registered {@link Converter} or <code>null</code> if not found  
831 \*/  
832 public Converter lookup(final Class<?> clazz) {  
833  
834 return (converters.get(clazz));  
835  
836 }  
837  
838 /\*\*  
839 \* Look up and return any registered {@link Converter} for the specified  
840 \* source and destination class; if there is no registered Converter,  
841 \* return <code>null</code>.  
842 \*  
843 \* @param sourceType Class of the value being converted  
844 \* @param targetType Class of the value to be converted to  
845 \* @return The registered {@link Converter} or <code>null</code> if not found  
846 \*/  
847 public Converter lookup(final Class<?> sourceType, final Class<?> targetType) {  
848  
849 if (targetType == null) {  
850 throw new IllegalArgumentException("Target type is missing");  
851 }  
852 if (sourceType == null) {  
853 return lookup(targetType);  
854 }  
855  
856 Converter converter = null;  
857 // Convert --> String  
858 if (targetType == String.class) {  
859 converter = lookup(sourceType);  
860 if (converter == null && (sourceType.isArray() ||  
861 Collection.class.isAssignableFrom(sourceType))) {  
862 converter = lookup(String[].class);  
863 }  
864 if (converter == null) {  
865 converter = lookup(String.class);  
866 }  
867 return converter;  
868 }  
869  
870 // Convert --> String array  
871 if (targetType == String[].class) {  
872 if (sourceType.isArray() || Collection.class.isAssignableFrom(sourceType)) {  
873 converter = lookup(sourceType);  
874 }  
875 if (converter == null) {  
876 converter = lookup(String[].class);  
877 }  
878 return converter;  
879 }  
880  
881 return lookup(targetType);  
882  
883 }  
884  
885 /\*\*  
886 \* Register a custom {@link Converter} for the specified destination  
887 \* <code>Class</code>, replacing any previously registered Converter.  
888 \*  
889 \* @param converter Converter to be registered  
890 \* @param clazz Destination class for conversions performed by this  
891 \* Converter  
892 \*/  
893 public void register(final Converter converter, final Class<?> clazz) {  
894  
895 converters.put(clazz, converter);  
896  
897 }  
898}