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.collections4;  
018  
019import java.util.Collection;  
020import java.util.Map;  
021  
022import org.apache.commons.collections4.functors.ChainedTransformer;  
023import org.apache.commons.collections4.functors.CloneTransformer;  
024import org.apache.commons.collections4.functors.ClosureTransformer;  
025import org.apache.commons.collections4.functors.ConstantTransformer;  
026import org.apache.commons.collections4.functors.EqualPredicate;  
027import org.apache.commons.collections4.functors.ExceptionTransformer;  
028import org.apache.commons.collections4.functors.FactoryTransformer;  
029import org.apache.commons.collections4.functors.IfTransformer;  
030import org.apache.commons.collections4.functors.InstantiateTransformer;  
031import org.apache.commons.collections4.functors.InvokerTransformer;  
032import org.apache.commons.collections4.functors.MapTransformer;  
033import org.apache.commons.collections4.functors.NOPTransformer;  
034import org.apache.commons.collections4.functors.PredicateTransformer;  
035import org.apache.commons.collections4.functors.StringValueTransformer;  
036import org.apache.commons.collections4.functors.SwitchTransformer;  
037  
038/\*\*  
039 \* <code>TransformerUtils</code> provides reference implementations and  
040 \* utilities for the Transformer functor interface. The supplied transformers are:  
041 \* <ul>  
042 \* <li>Invoker - returns the result of a method call on the input object  
043 \* <li>Clone - returns a clone of the input object  
044 \* <li>Constant - always returns the same object  
045 \* <li>Closure - performs a Closure and returns the input object  
046 \* <li>Predicate - returns the result of the predicate as a Boolean  
047 \* <li>Factory - returns a new object from a factory  
048 \* <li>Chained - chains two or more transformers together  
049 \* <li>If - calls one transformer or another based on a predicate  
050 \* <li>Switch - calls one transformer based on one or more predicates  
051 \* <li>SwitchMap - calls one transformer looked up from a Map  
052 \* <li>Instantiate - the Class input object is instantiated  
053 \* <li>Map - returns an object from a supplied Map  
054 \* <li>Null - always returns null  
055 \* <li>NOP - returns the input object, which should be immutable  
056 \* <li>Exception - always throws an exception  
057 \* <li>StringValue - returns a <code>java.lang.String</code> representation of the input object  
058 \* </ul>  
059 \* <p>  
060 \* Since v4.1 only transformers which are considered to be safe are  
061 \* Serializable. Transformers considered to be unsafe for serialization are:  
062 \* </p>  
063 \* <ul>  
064 \* <li>Invoker  
065 \* <li>Clone  
066 \* <li>Instantiate  
067 \* </ul>  
068 \*  
069 \* @since 3.0  
070 \*/  
071public class TransformerUtils {  
072  
073 /\*\*  
074 \* This class is not normally instantiated.  
075 \*/  
076 private TransformerUtils() {}  
077  
078 /\*\*  
079 \* Gets a transformer that always throws an exception.  
080 \* This could be useful during testing as a placeholder.  
081 \*  
082 \* @param <I> the input type  
083 \* @param <O> the output type  
084 \* @return the transformer  
085 \* @see ExceptionTransformer  
086 \*/  
087 public static <I, O> Transformer<I, O> exceptionTransformer() {  
088 return ExceptionTransformer.exceptionTransformer();  
089 }  
090  
091 /\*\*  
092 \* Gets a transformer that always returns null.  
093 \*  
094 \* @param <I> the input type  
095 \* @param <O> the output type  
096 \* @return the transformer  
097 \* @see ConstantTransformer  
098 \*/  
099 public static <I, O> Transformer<I, O> nullTransformer() {  
100 return ConstantTransformer.nullTransformer();  
101 }  
102  
103 /\*\*  
104 \* Gets a transformer that returns the input object.  
105 \* The input object should be immutable to maintain the  
106 \* contract of Transformer (although this is not checked).  
107 \*  
108 \* @param <T> the input/output type  
109 \* @return the transformer  
110 \* @see NOPTransformer  
111 \*/  
112 public static <T> Transformer<T, T> nopTransformer() {  
113 return NOPTransformer.nopTransformer();  
114 }  
115  
116 /\*\*  
117 \* Gets a transformer that returns a clone of the input object.  
118 \* The input object will be cloned using one of these techniques (in order):  
119 \* <ul>  
120 \* <li>public clone method</li>  
121 \* <li>public copy constructor</li>  
122 \* <li>serialization clone</li>  
123 \* </ul>  
124 \*  
125 \* @param <T> the input/output type  
126 \* @return the transformer  
127 \* @see CloneTransformer  
128 \*/  
129 public static <T> Transformer<T, T> cloneTransformer() {  
130 return CloneTransformer.cloneTransformer();  
131 }  
132  
133 /\*\*  
134 \* Creates a Transformer that will return the same object each time the  
135 \* transformer is used.  
136 \*  
137 \* @param <I> the input type  
138 \* @param <O> the output type  
139 \* @param constantToReturn the constant object to return each time in the transformer  
140 \* @return the transformer.  
141 \* @see ConstantTransformer  
142 \*/  
143 public static <I, O> Transformer<I, O> constantTransformer(final O constantToReturn) {  
144 return ConstantTransformer.constantTransformer(constantToReturn);  
145 }  
146  
147 /\*\*  
148 \* Creates a Transformer that calls a Closure each time the transformer is used.  
149 \* The transformer returns the input object.  
150 \*  
151 \* @param <T> the input/output type  
152 \* @param closure the closure to run each time in the transformer, not null  
153 \* @return the transformer  
154 \* @throws NullPointerException if the closure is null  
155 \* @see ClosureTransformer  
156 \*/  
157 public static <T> Transformer<T, T> asTransformer(final Closure<? super T> closure) {  
158 return ClosureTransformer.closureTransformer(closure);  
159 }  
160  
161 /\*\*  
162 \* Creates a Transformer that calls a Predicate each time the transformer is used.  
163 \* The transformer will return either Boolean.TRUE or Boolean.FALSE.  
164 \*  
165 \* @param <T> the input type  
166 \* @param predicate the predicate to run each time in the transformer, not null  
167 \* @return the transformer  
168 \* @throws NullPointerException if the predicate is null  
169 \* @see PredicateTransformer  
170 \*/  
171 public static <T> Transformer<T, Boolean> asTransformer(final Predicate<? super T> predicate) {  
172 return PredicateTransformer.predicateTransformer(predicate);  
173 }  
174  
175 /\*\*  
176 \* Creates a Transformer that calls a Factory each time the transformer is used.  
177 \* The transformer will return the value returned by the factory.  
178 \*  
179 \* @param <I> the input type  
180 \* @param <O> the output type  
181 \* @param factory the factory to run each time in the transformer, not null  
182 \* @return the transformer  
183 \* @throws NullPointerException if the factory is null  
184 \* @see FactoryTransformer  
185 \*/  
186 public static <I, O> Transformer<I, O> asTransformer(final Factory<? extends O> factory) {  
187 return FactoryTransformer.factoryTransformer(factory);  
188 }  
189  
190 /\*\*  
191 \* Create a new Transformer that calls each transformer in turn, passing the  
192 \* result into the next transformer.  
193 \*  
194 \* @param <T> the input/output type  
195 \* @param transformers an array of transformers to chain  
196 \* @return the transformer  
197 \* @throws NullPointerException if the transformers array or any of the transformers is null  
198 \* @see ChainedTransformer  
199 \*/  
200 public static <T> Transformer<T, T> chainedTransformer(  
201 final Transformer<? super T, ? extends T>... transformers) {  
202 return ChainedTransformer.chainedTransformer(transformers);  
203 }  
204  
205 /\*\*  
206 \* Create a new Transformer that calls each transformer in turn, passing the  
207 \* result into the next transformer. The ordering is that of the iterator()  
208 \* method on the collection.  
209 \*  
210 \* @param <T> the input/output type  
211 \* @param transformers a collection of transformers to chain  
212 \* @return the transformer  
213 \* @throws NullPointerException if the transformers collection or any of the transformers is null  
214 \* @see ChainedTransformer  
215 \*/  
216 public static <T> Transformer<T, T> chainedTransformer(  
217 final Collection<? extends Transformer<? super T, ? extends T>> transformers) {  
218 return ChainedTransformer.chainedTransformer(transformers);  
219 }  
220  
221 /\*\*  
222 \* Create a new Transformer that calls the transformer if the predicate is true,  
223 \* otherwise the input object is returned unchanged.  
224 \*  
225 \* @param <T> the input / output type  
226 \* @param predicate the predicate to switch on  
227 \* @param trueTransformer the transformer called if the predicate is true  
228 \* @return the transformer  
229 \* @throws NullPointerException if either the predicate or transformer is null  
230 \* @see IfTransformer  
231 \* @since 4.1  
232 \*/  
233 public static <T> Transformer<T, T> ifTransformer(final Predicate<? super T> predicate,  
234 final Transformer<? super T, ? extends T> trueTransformer) {  
235 return IfTransformer.ifTransformer(predicate, trueTransformer);  
236 }  
237  
238 /\*\*  
239 \* Create a new Transformer that calls one of two transformers depending  
240 \* on the specified predicate.  
241 \*  
242 \* @param <I> the input type  
243 \* @param <O> the output type  
244 \* @param predicate the predicate to switch on  
245 \* @param trueTransformer the transformer called if the predicate is true  
246 \* @param falseTransformer the transformer called if the predicate is false  
247 \* @return the transformer  
248 \* @throws NullPointerException if either the predicate or transformer is null  
249 \* @see IfTransformer  
250 \* @since 4.1  
251 \*/  
252 public static <I, O> Transformer<I, O> ifTransformer(final Predicate<? super I> predicate,  
253 final Transformer<? super I, ? extends O> trueTransformer,  
254 final Transformer<? super I, ? extends O> falseTransformer) {  
255 return IfTransformer.ifTransformer(predicate, trueTransformer, falseTransformer);  
256 }  
257  
258 /\*\*  
259 \* Create a new Transformer that calls one of two transformers depending  
260 \* on the specified predicate.  
261 \*  
262 \* @param <I> the input type  
263 \* @param <O> the output type  
264 \* @param predicate the predicate to switch on  
265 \* @param trueTransformer the transformer called if the predicate is true  
266 \* @param falseTransformer the transformer called if the predicate is false  
267 \* @return the transformer  
268 \* @throws NullPointerException if either the predicate or transformer is null  
269 \* @see SwitchTransformer  
270 \* @deprecated as of 4.1, use {@link #ifTransformer(Predicate, Transformer, Transformer)}  
271 \*/  
272 @SuppressWarnings("unchecked")  
273 @Deprecated  
274 public static <I, O> Transformer<I, O> switchTransformer(final Predicate<? super I> predicate,  
275 final Transformer<? super I, ? extends O> trueTransformer,  
276 final Transformer<? super I, ? extends O> falseTransformer) {  
277 return SwitchTransformer.switchTransformer(new Predicate[] { predicate },  
278 new Transformer[] { trueTransformer }, falseTransformer);  
279 }  
280  
281 /\*\*  
282 \* Create a new Transformer that calls one of the transformers depending  
283 \* on the predicates. The transformer at array location 0 is called if the  
284 \* predicate at array location 0 returned true. Each predicate is evaluated  
285 \* until one returns true. If no predicates evaluate to true, null is returned.  
286 \*  
287 \* @param <I> the input type  
288 \* @param <O> the output type  
289 \* @param predicates an array of predicates to check  
290 \* @param transformers an array of transformers to call  
291 \* @return the transformer  
292 \* @throws NullPointerException if the either array is null  
293 \* @throws NullPointerException if any element in the arrays is null  
294 \* @throws IllegalArgumentException if the arrays have different sizes  
295 \* @see SwitchTransformer  
296 \*/  
297 public static <I, O> Transformer<I, O> switchTransformer(final Predicate<? super I>[] predicates,  
298 final Transformer<? super I, ? extends O>[] transformers) {  
299 return SwitchTransformer.switchTransformer(predicates, transformers, null);  
300 }  
301  
302 /\*\*  
303 \* Create a new Transformer that calls one of the transformers depending  
304 \* on the predicates. The transformer at array location 0 is called if the  
305 \* predicate at array location 0 returned true. Each predicate is evaluated  
306 \* until one returns true. If no predicates evaluate to true, the default  
307 \* transformer is called. If the default transformer is null, null is returned.  
308 \*  
309 \* @param <I> the input type  
310 \* @param <O> the output type  
311 \* @param predicates an array of predicates to check  
312 \* @param transformers an array of transformers to call  
313 \* @param defaultTransformer the default to call if no predicate matches, null means return null  
314 \* @return the transformer  
315 \* @throws NullPointerException if the either array is null  
316 \* @throws NullPointerException if any element in the arrays is null  
317 \* @throws IllegalArgumentException if the arrays have different sizes  
318 \* @see SwitchTransformer  
319 \*/  
320 public static <I, O> Transformer<I, O> switchTransformer(final Predicate<? super I>[] predicates,  
321 final Transformer<? super I, ? extends O>[] transformers,  
322 final Transformer<? super I, ? extends O> defaultTransformer) {  
323 return SwitchTransformer.switchTransformer(predicates, transformers, defaultTransformer);  
324 }  
325  
326 /\*\*  
327 \* Create a new Transformer that calls one of the transformers depending  
328 \* on the predicates.  
329 \* <p>  
330 \* The Map consists of Predicate keys and Transformer values. A transformer  
331 \* is called if its matching predicate returns true. Each predicate is evaluated  
332 \* until one returns true. If no predicates evaluate to true, the default  
333 \* transformer is called. The default transformer is set in the map with a  
334 \* null key. If no default transformer is set, null will be returned in a default  
335 \* case. The ordering is that of the iterator() method on the entryset collection  
336 \* of the map.  
337 \*  
338 \* @param <I> the input type  
339 \* @param <O> the output type  
340 \* @param predicatesAndTransformers a map of predicates to transformers  
341 \* @return the transformer  
342 \* @throws NullPointerException if the map is null  
343 \* @throws NullPointerException if any transformer in the map is null  
344 \* @throws ClassCastException if the map elements are of the wrong type  
345 \* @see SwitchTransformer  
346 \*/  
347 public static <I, O> Transformer<I, O> switchTransformer(  
348 final Map<Predicate<I>, Transformer<I, O>> predicatesAndTransformers) {  
349 return SwitchTransformer.switchTransformer(predicatesAndTransformers);  
350 }  
351  
352 /\*\*  
353 \* Create a new Transformer that uses the input object as a key to find the  
354 \* transformer to call.  
355 \* <p>  
356 \* The Map consists of object keys and Transformer values. A transformer  
357 \* is called if the input object equals the key. If there is no match, the  
358 \* default transformer is called. The default transformer is set in the map  
359 \* using a null key. If no default is set, null will be returned in a default case.  
360 \*  
361 \* @param <I> the input type  
362 \* @param <O> the output type  
363 \* @param objectsAndTransformers a map of objects to transformers  
364 \* @return the transformer  
365 \* @throws NullPointerException if the map is null  
366 \* @throws NullPointerException if any transformer in the map is null  
367 \* @see SwitchTransformer  
368 \*/  
369 @SuppressWarnings("unchecked")  
370 public static <I, O> Transformer<I, O> switchMapTransformer(  
371 final Map<I, Transformer<I, O>> objectsAndTransformers) {  
372  
373 if (objectsAndTransformers == null) {  
374 throw new NullPointerException("The object and transformer map must not be null");  
375 }  
376 final Transformer<? super I, ? extends O> def = objectsAndTransformers.remove(null);  
377 final int size = objectsAndTransformers.size();  
378 final Transformer<? super I, ? extends O>[] trs = new Transformer[size];  
379 final Predicate<I>[] preds = new Predicate[size];  
380 int i = 0;  
381 for (final Map.Entry<I, Transformer<I, O>> entry : objectsAndTransformers.entrySet()) {  
382 preds[i] = EqualPredicate.<I>equalPredicate(entry.getKey());  
383 trs[i++] = entry.getValue();  
384 }  
385 return TransformerUtils.switchTransformer(preds, trs, def);  
386 }  
387  
388 /\*\*  
389 \* Gets a Transformer that expects an input Class object that it will instantiate.  
390 \*  
391 \* @param <T> the output type  
392 \* @return the transformer  
393 \* @see InstantiateTransformer  
394 \*/  
395 public static <T> Transformer<Class<? extends T>, T> instantiateTransformer() {  
396 return InstantiateTransformer.instantiateTransformer();  
397 }  
398  
399 /\*\*  
400 \* Creates a Transformer that expects an input Class object that it will  
401 \* instantiate. The constructor used is determined by the arguments specified  
402 \* to this method.  
403 \*  
404 \* @param <T> the output type  
405 \* @param paramTypes parameter types for the constructor, can be null  
406 \* @param args the arguments to pass to the constructor, can be null  
407 \* @return the transformer  
408 \* @throws IllegalArgumentException if the paramTypes and args don't match  
409 \* @see InstantiateTransformer  
410 \*/  
411 public static <T> Transformer<Class<? extends T>, T> instantiateTransformer(  
412 final Class<?>[] paramTypes, final Object[] args) {  
413 return InstantiateTransformer.instantiateTransformer(paramTypes, args);  
414 }  
415  
416 /\*\*  
417 \* Creates a Transformer that uses the passed in Map to transform the input  
418 \* object (as a simple lookup).  
419 \*  
420 \* @param <I> the input type  
421 \* @param <O> the output type  
422 \* @param map the map to use to transform the objects  
423 \* @return the transformer, or {@link ConstantTransformer#nullTransformer()} if the  
424 \* {@code map} is {@code null}  
425 \* @see MapTransformer  
426 \*/  
427 public static <I, O> Transformer<I, O> mapTransformer(final Map<? super I, ? extends O> map) {  
428 return MapTransformer.mapTransformer(map);  
429 }  
430  
431 /\*\*  
432 \* Gets a Transformer that invokes a method on the input object.  
433 \* The method must have no parameters. If the input object is {@code null},  
434 \* {@code null} is returned.  
435 \*  
436 \* <p>  
437 \* For example, <code>TransformerUtils.invokerTransformer("getName");</code>  
438 \* will call the <code>getName</code> method on the input object to  
439 \* determine the transformer result.  
440 \* </p>  
441 \*  
442 \* @param <I> the input type  
443 \* @param <O> the output type  
444 \* @param methodName the method name to call on the input object, may not be null  
445 \* @return the transformer  
446 \* @throws NullPointerException if the methodName is null.  
447 \* @see InvokerTransformer  
448 \*/  
449 public static <I, O> Transformer<I, O> invokerTransformer(final String methodName) {  
450 return InvokerTransformer.invokerTransformer(methodName, null, null);  
451 }  
452  
453 /\*\*  
454 \* Gets a Transformer that invokes a method on the input object.  
455 \* The method parameters are specified. If the input object is {@code null},  
456 \* {@code null} is returned.  
457 \*  
458 \* @param <I> the input type  
459 \* @param <O> the output type  
460 \* @param methodName the name of the method  
461 \* @param paramTypes the parameter types  
462 \* @param args the arguments  
463 \* @return the transformer  
464 \* @throws NullPointerException if the method name is null  
465 \* @throws IllegalArgumentException if the paramTypes and args don't match  
466 \* @see InvokerTransformer  
467 \*/  
468 public static <I, O> Transformer<I, O> invokerTransformer(final String methodName, final Class<?>[] paramTypes,  
469 final Object[] args) {  
470 return InvokerTransformer.invokerTransformer(methodName, paramTypes, args);  
471 }  
472  
473 /\*\*  
474 \* Gets a transformer that returns a <code>java.lang.String</code>  
475 \* representation of the input object. This is achieved via the  
476 \* <code>toString</code> method, <code>null</code> returns 'null'.  
477 \*  
478 \* @param <T> the input type  
479 \* @return the transformer  
480 \* @see StringValueTransformer  
481 \*/  
482 public static <T> Transformer<T, String> stringValueTransformer() {  
483 return StringValueTransformer.stringValueTransformer();  
484 }  
485  
486}