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.list;  
018  
019import java.util.Collection;  
020import java.util.List;  
021import java.util.ListIterator;  
022  
023import org.apache.commons.collections4.Transformer;  
024import org.apache.commons.collections4.collection.TransformedCollection;  
025import org.apache.commons.collections4.iterators.AbstractListIteratorDecorator;  
026  
027/\*\*  
028 \* Decorates another <code>List</code> to transform objects that are added.  
029 \* <p>  
030 \* The add and set methods are affected by this class.  
031 \* Thus objects must be removed or searched for using their transformed form.  
032 \* For example, if the transformation converts Strings to Integers, you must  
033 \* use the Integer form to remove objects.  
034 \* </p>  
035 \* <p>  
036 \* This class is Serializable from Commons Collections 3.1.  
037 \* </p>  
038 \*  
039 \* @since 3.0  
040 \*/  
041public class TransformedList<E> extends TransformedCollection<E> implements List<E> {  
042  
043 /\*\* Serialization version \*/  
044 private static final long serialVersionUID = 1077193035000013141L;  
045  
046 /\*\*  
047 \* Factory method to create a transforming list.  
048 \* <p>  
049 \* If there are any elements already in the list being decorated, they  
050 \* are NOT transformed.  
051 \* Contrast this with {@link #transformedList(List, Transformer)}.  
052 \*  
053 \* @param <E> the type of the elements in the list  
054 \* @param list the list to decorate, must not be null  
055 \* @param transformer the transformer to use for conversion, must not be null  
056 \* @return a new transformed list  
057 \* @throws NullPointerException if list or transformer is null  
058 \* @since 4.0  
059 \*/  
060 public static <E> TransformedList<E> transformingList(final List<E> list,  
061 final Transformer<? super E, ? extends E> transformer) {  
062 return new TransformedList<>(list, transformer);  
063 }  
064  
065 /\*\*  
066 \* Factory method to create a transforming list that will transform  
067 \* existing contents of the specified list.  
068 \* <p>  
069 \* If there are any elements already in the list being decorated, they  
070 \* will be transformed by this method.  
071 \* Contrast this with {@link #transformingList(List, Transformer)}.  
072 \*  
073 \* @param <E> the type of the elements in the list  
074 \* @param list the list to decorate, must not be null  
075 \* @param transformer the transformer to use for conversion, must not be null  
076 \* @return a new transformed List  
077 \* @throws NullPointerException if list or transformer is null  
078 \* @since 4.0  
079 \*/  
080 public static <E> TransformedList<E> transformedList(final List<E> list,  
081 final Transformer<? super E, ? extends E> transformer) {  
082 final TransformedList<E> decorated = new TransformedList<>(list, transformer);  
083 if (list.size() > 0) {  
084 @SuppressWarnings("unchecked") // list is of type E  
085 final E[] values = (E[]) list.toArray(); // NOPMD - false positive for generics  
086 list.clear();  
087 for (final E value : values) {  
088 decorated.decorated().add(transformer.transform(value));  
089 }  
090 }  
091 return decorated;  
092 }  
093  
094 //-----------------------------------------------------------------------  
095 /\*\*  
096 \* Constructor that wraps (not copies).  
097 \* <p>  
098 \* If there are any elements already in the list being decorated, they  
099 \* are NOT transformed.  
100 \*  
101 \* @param list the list to decorate, must not be null  
102 \* @param transformer the transformer to use for conversion, must not be null  
103 \* @throws NullPointerException if list or transformer is null  
104 \*/  
105 protected TransformedList(final List<E> list, final Transformer<? super E, ? extends E> transformer) {  
106 super(list, transformer);  
107 }  
108  
109 /\*\*  
110 \* Gets the decorated list.  
111 \*  
112 \* @return the decorated list  
113 \*/  
114 protected List<E> getList() {  
115 return (List<E>) decorated();  
116 }  
117  
118 @Override  
119 public boolean equals(final Object object) {  
120 return object == this || decorated().equals(object);  
121 }  
122  
123 @Override  
124 public int hashCode() {  
125 return decorated().hashCode();  
126 }  
127  
128 //-----------------------------------------------------------------------  
129  
130 @Override  
131 public E get(final int index) {  
132 return getList().get(index);  
133 }  
134  
135 @Override  
136 public int indexOf(final Object object) {  
137 return getList().indexOf(object);  
138 }  
139  
140 @Override  
141 public int lastIndexOf(final Object object) {  
142 return getList().lastIndexOf(object);  
143 }  
144  
145 @Override  
146 public E remove(final int index) {  
147 return getList().remove(index);  
148 }  
149  
150 //-----------------------------------------------------------------------  
151  
152 @Override  
153 public void add(final int index, E object) {  
154 object = transform(object);  
155 getList().add(index, object);  
156 }  
157  
158 @Override  
159 public boolean addAll(final int index, Collection<? extends E> coll) {  
160 coll = transform(coll);  
161 return getList().addAll(index, coll);  
162 }  
163  
164 @Override  
165 public ListIterator<E> listIterator() {  
166 return listIterator(0);  
167 }  
168  
169 @Override  
170 public ListIterator<E> listIterator(final int i) {  
171 return new TransformedListIterator(getList().listIterator(i));  
172 }  
173  
174 @Override  
175 public E set(final int index, E object) {  
176 object = transform(object);  
177 return getList().set(index, object);  
178 }  
179  
180 @Override  
181 public List<E> subList(final int fromIndex, final int toIndex) {  
182 final List<E> sub = getList().subList(fromIndex, toIndex);  
183 return new TransformedList<>(sub, transformer);  
184 }  
185  
186 /\*\*  
187 \* Inner class Iterator for the TransformedList  
188 \*/  
189 protected class TransformedListIterator extends AbstractListIteratorDecorator<E> {  
190  
191 /\*\*  
192 \* Create a new transformed list iterator.  
193 \*  
194 \* @param iterator the list iterator to decorate  
195 \*/  
196 protected TransformedListIterator(final ListIterator<E> iterator) {  
197 super(iterator);  
198 }  
199  
200 @Override  
201 public void add(E object) {  
202 object = transform(object);  
203 getListIterator().add(object);  
204 }  
205  
206 @Override  
207 public void set(E object) {  
208 object = transform(object);  
209 getListIterator().set(object);  
210 }  
211 }  
212  
213}