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.map;  
018  
019import java.io.IOException;  
020import java.io.ObjectInputStream;  
021import java.io.ObjectOutputStream;  
022import java.io.Serializable;  
023import java.util.AbstractList;  
024import java.util.Collection;  
025import java.util.Iterator;  
026import java.util.List;  
027import java.util.ListIterator;  
028import java.util.Map;  
029import java.util.function.Predicate;  
030  
031import org.apache.commons.collections4.iterators.UnmodifiableIterator;  
032import org.apache.commons.collections4.iterators.UnmodifiableListIterator;  
033import org.apache.commons.collections4.list.UnmodifiableList;  
034  
035/\*\*  
036 \* A <code>Map</code> implementation that maintains the order of the entries.  
037 \* In this implementation order is maintained by original insertion.  
038 \* <p>  
039 \* This implementation improves on the JDK1.4 LinkedHashMap by adding the  
040 \* {@link org.apache.commons.collections4.MapIterator MapIterator}  
041 \* functionality, additional convenience methods and allowing  
042 \* bidirectional iteration. It also implements <code>OrderedMap</code>.  
043 \* In addition, non-interface methods are provided to access the map by index.  
044 \* </p>  
045 \* <p>  
046 \* The <code>orderedMapIterator()</code> method provides direct access to a  
047 \* bidirectional iterator. The iterators from the other views can also be cast  
048 \* to <code>OrderedIterator</code> if required.  
049 \* </p>  
050 \* <p>  
051 \* All the available iterators can be reset back to the start by casting to  
052 \* <code>ResettableIterator</code> and calling <code>reset()</code>.  
053 \* </p>  
054 \* <p>  
055 \* The implementation is also designed to be subclassed, with lots of useful  
056 \* methods exposed.  
057 \* </p>  
058 \* <p>  
059 \* <strong>Note that LinkedMap is not synchronized and is not thread-safe.</strong>  
060 \* If you wish to use this map from multiple threads concurrently, you must use  
061 \* appropriate synchronization. The simplest approach is to wrap this map  
062 \* using {@link java.util.Collections#synchronizedMap(Map)}. This class may throw  
063 \* exceptions when accessed by concurrent threads without synchronization.  
064 \* </p>  
065 \*  
066 \* @param <K> the type of the keys in this map  
067 \* @param <V> the type of the values in this map  
068 \* @since 3.0  
069 \*/  
070public class LinkedMap<K, V> extends AbstractLinkedMap<K, V> implements Serializable, Cloneable {  
071  
072 /\*\* Serialisation version \*/  
073 private static final long serialVersionUID = 9077234323521161066L;  
074  
075 /\*\*  
076 \* Constructs a new empty map with default size and load factor.  
077 \*/  
078 public LinkedMap() {  
079 super(DEFAULT\_CAPACITY, DEFAULT\_LOAD\_FACTOR, DEFAULT\_THRESHOLD);  
080 }  
081  
082 /\*\*  
083 \* Constructs a new, empty map with the specified initial capacity.  
084 \*  
085 \* @param initialCapacity the initial capacity  
086 \* @throws IllegalArgumentException if the initial capacity is negative  
087 \*/  
088 public LinkedMap(final int initialCapacity) {  
089 super(initialCapacity);  
090 }  
091  
092 /\*\*  
093 \* Constructs a new, empty map with the specified initial capacity and  
094 \* load factor.  
095 \*  
096 \* @param initialCapacity the initial capacity  
097 \* @param loadFactor the load factor  
098 \* @throws IllegalArgumentException if the initial capacity is negative  
099 \* @throws IllegalArgumentException if the load factor is less than zero  
100 \*/  
101 public LinkedMap(final int initialCapacity, final float loadFactor) {  
102 super(initialCapacity, loadFactor);  
103 }  
104  
105 /\*\*  
106 \* Constructor copying elements from another map.  
107 \*  
108 \* @param map the map to copy  
109 \* @throws NullPointerException if the map is null  
110 \*/  
111 public LinkedMap(final Map<? extends K, ? extends V> map) {  
112 super(map);  
113 }  
114  
115 //-----------------------------------------------------------------------  
116 /\*\*  
117 \* Clones the map without cloning the keys or values.  
118 \*  
119 \* @return a shallow clone  
120 \*/  
121 @Override  
122 public LinkedMap<K, V> clone() {  
123 return (LinkedMap<K, V>) super.clone();  
124 }  
125  
126 /\*\*  
127 \* Write the map out using a custom routine.  
128 \*  
129 \* @param out the output stream  
130 \* @throws IOException if an error occurs while writing to the stream  
131 \*/  
132 private void writeObject(final ObjectOutputStream out) throws IOException {  
133 out.defaultWriteObject();  
134 doWriteObject(out);  
135 }  
136  
137 /\*\*  
138 \* Read the map in using a custom routine.  
139 \*  
140 \* @param in the input stream  
141 \* @throws IOException if an error occurs while reading from the stream  
142 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
143 \*/  
144 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
145 in.defaultReadObject();  
146 doReadObject(in);  
147 }  
148  
149 //-----------------------------------------------------------------------  
150 /\*\*  
151 \* Gets the key at the specified index.  
152 \*  
153 \* @param index the index to retrieve  
154 \* @return the key at the specified index  
155 \* @throws IndexOutOfBoundsException if the index is invalid  
156 \*/  
157 public K get(final int index) {  
158 return getEntry(index).getKey();  
159 }  
160  
161 /\*\*  
162 \* Gets the value at the specified index.  
163 \*  
164 \* @param index the index to retrieve  
165 \* @return the value at the specified index  
166 \* @throws IndexOutOfBoundsException if the index is invalid  
167 \*/  
168 public V getValue(final int index) {  
169 return getEntry(index).getValue();  
170 }  
171  
172 /\*\*  
173 \* Gets the index of the specified key.  
174 \*  
175 \* @param key the key to find the index of  
176 \* @return the index, or -1 if not found  
177 \*/  
178 public int indexOf(Object key) {  
179 key = convertKey(key);  
180 int i = 0;  
181 for (LinkEntry<K, V> entry = header.after; entry != header; entry = entry.after, i++) {  
182 if (isEqualKey(key, entry.key)) {  
183 return i;  
184 }  
185 }  
186 return -1;  
187 }  
188  
189 /\*\*  
190 \* Removes the element at the specified index.  
191 \*  
192 \* @param index the index of the object to remove  
193 \* @return the previous value corresponding the <code>key</code>,  
194 \* or <code>null</code> if none existed  
195 \* @throws IndexOutOfBoundsException if the index is invalid  
196 \*/  
197 public V remove(final int index) {  
198 return remove(get(index));  
199 }  
200  
201 /\*\*  
202 \* Gets an unmodifiable List view of the keys.  
203 \* <p>  
204 \* The returned list is unmodifiable because changes to the values of  
205 \* the list (using {@link java.util.ListIterator#set(Object)}) will  
206 \* effectively remove the value from the list and reinsert that value at  
207 \* the end of the list, which is an unexpected side effect of changing the  
208 \* value of a list. This occurs because changing the key, changes when the  
209 \* mapping is added to the map and thus where it appears in the list.  
210 \* <p>  
211 \* An alternative to this method is to use {@link #keySet()}.  
212 \*  
213 \* @see #keySet()  
214 \* @return The ordered list of keys.  
215 \*/  
216 public List<K> asList() {  
217 return new LinkedMapList<>(this);  
218 }  
219  
220 /\*\*  
221 \* List view of map.  
222 \*/  
223 static class LinkedMapList<K> extends AbstractList<K> {  
224  
225 private final LinkedMap<K, ?> parent;  
226  
227 LinkedMapList(final LinkedMap<K, ?> parent) {  
228 this.parent = parent;  
229 }  
230  
231 @Override  
232 public int size() {  
233 return parent.size();  
234 }  
235  
236 @Override  
237 public K get(final int index) {  
238 return parent.get(index);  
239 }  
240  
241 @Override  
242 public boolean contains(final Object obj) {  
243 return parent.containsKey(obj);  
244 }  
245  
246 @Override  
247 public int indexOf(final Object obj) {  
248 return parent.indexOf(obj);  
249 }  
250  
251 @Override  
252 public int lastIndexOf(final Object obj) {  
253 return parent.indexOf(obj);  
254 }  
255  
256 @Override  
257 public boolean containsAll(final Collection<?> coll) {  
258 return parent.keySet().containsAll(coll);  
259 }  
260  
261 @Override  
262 public K remove(final int index) {  
263 throw new UnsupportedOperationException();  
264 }  
265  
266 @Override  
267 public boolean remove(final Object obj) {  
268 throw new UnsupportedOperationException();  
269 }  
270  
271 /\*\*  
272 \* @since 4.4  
273 \*/  
274 @Override  
275 public boolean removeIf(final Predicate<? super K> filter) {  
276 throw new UnsupportedOperationException();  
277 }  
278  
279 @Override  
280 public boolean removeAll(final Collection<?> coll) {  
281 throw new UnsupportedOperationException();  
282 }  
283  
284 @Override  
285 public boolean retainAll(final Collection<?> coll) {  
286 throw new UnsupportedOperationException();  
287 }  
288  
289 @Override  
290 public void clear() {  
291 throw new UnsupportedOperationException();  
292 }  
293  
294 @Override  
295 public Object[] toArray() {  
296 return parent.keySet().toArray();  
297 }  
298  
299 @Override  
300 public <T> T[] toArray(final T[] array) {  
301 return parent.keySet().toArray(array);  
302 }  
303  
304 @Override  
305 public Iterator<K> iterator() {  
306 return UnmodifiableIterator.unmodifiableIterator(parent.keySet().iterator());  
307 }  
308  
309 @Override  
310 public ListIterator<K> listIterator() {  
311 return UnmodifiableListIterator.umodifiableListIterator(super.listIterator());  
312 }  
313  
314 @Override  
315 public ListIterator<K> listIterator(final int fromIndex) {  
316 return UnmodifiableListIterator.umodifiableListIterator(super.listIterator(fromIndex));  
317 }  
318  
319 @Override  
320 public List<K> subList(final int fromIndexInclusive, final int toIndexExclusive) {  
321 return UnmodifiableList.unmodifiableList(super.subList(fromIndexInclusive, toIndexExclusive));  
322 }  
323 }  
324  
325}