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.iterators;  
018  
019import java.util.Iterator;  
020import java.util.Map;  
021  
022import org.apache.commons.collections4.MapIterator;  
023import org.apache.commons.collections4.ResettableIterator;  
024  
025/\*\*  
026 \* Implements a <code>MapIterator</code> using a Map entrySet.  
027 \* Reverse iteration is not supported.  
028 \* <pre>  
029 \* MapIterator it = map.mapIterator();  
030 \* while (it.hasNext()) {  
031 \* Object key = it.next();  
032 \* Object value = it.getValue();  
033 \* it.setValue(newValue);  
034 \* }  
035 \* </pre>  
036 \*  
037 \* @param <K> the type of keys  
038 \* @param <V> the type of mapped values  
039 \* @since 3.0  
040 \*/  
041public class EntrySetMapIterator<K, V> implements MapIterator<K, V>, ResettableIterator<K> {  
042  
043 private final Map<K, V> map;  
044 private Iterator<Map.Entry<K, V>> iterator;  
045 private Map.Entry<K, V> last;  
046 private boolean canRemove = false;  
047  
048 /\*\*  
049 \* Constructor.  
050 \*  
051 \* @param map the map to iterate over  
052 \*/  
053 public EntrySetMapIterator(final Map<K, V> map) {  
054 super();  
055 this.map = map;  
056 this.iterator = map.entrySet().iterator();  
057 }  
058  
059 //-----------------------------------------------------------------------  
060 /\*\*  
061 \* Checks to see if there are more entries still to be iterated.  
062 \*  
063 \* @return <code>true</code> if the iterator has more elements  
064 \*/  
065 @Override  
066 public boolean hasNext() {  
067 return iterator.hasNext();  
068 }  
069  
070 /\*\*  
071 \* Gets the next <em>key</em> from the <code>Map</code>.  
072 \*  
073 \* @return the next key in the iteration  
074 \* @throws java.util.NoSuchElementException if the iteration is finished  
075 \*/  
076 @Override  
077 public K next() {  
078 last = iterator.next();  
079 canRemove = true;  
080 return last.getKey();  
081 }  
082  
083 //-----------------------------------------------------------------------  
084 /\*\*  
085 \* Removes the last returned key from the underlying <code>Map</code>.  
086 \* <p>  
087 \* This method can be called once per call to <code>next()</code>.  
088 \*  
089 \* @throws UnsupportedOperationException if remove is not supported by the map  
090 \* @throws IllegalStateException if <code>next()</code> has not yet been called  
091 \* @throws IllegalStateException if <code>remove()</code> has already been called  
092 \* since the last call to <code>next()</code>  
093 \*/  
094 @Override  
095 public void remove() {  
096 if (canRemove == false) {  
097 throw new IllegalStateException("Iterator remove() can only be called once after next()");  
098 }  
099 iterator.remove();  
100 last = null;  
101 canRemove = false;  
102 }  
103  
104 //-----------------------------------------------------------------------  
105 /\*\*  
106 \* Gets the current key, which is the key returned by the last call  
107 \* to <code>next()</code>.  
108 \*  
109 \* @return the current key  
110 \* @throws IllegalStateException if <code>next()</code> has not yet been called  
111 \*/  
112 @Override  
113 public K getKey() {  
114 if (last == null) {  
115 throw new IllegalStateException("Iterator getKey() can only be called after next() and before remove()");  
116 }  
117 return last.getKey();  
118 }  
119  
120 /\*\*  
121 \* Gets the current value, which is the value associated with the last key  
122 \* returned by <code>next()</code>.  
123 \*  
124 \* @return the current value  
125 \* @throws IllegalStateException if <code>next()</code> has not yet been called  
126 \*/  
127 @Override  
128 public V getValue() {  
129 if (last == null) {  
130 throw new IllegalStateException("Iterator getValue() can only be called after next() and before remove()");  
131 }  
132 return last.getValue();  
133 }  
134  
135 /\*\*  
136 \* Sets the value associated with the current key.  
137 \*  
138 \* @param value the new value  
139 \* @return the previous value  
140 \* @throws UnsupportedOperationException if setValue is not supported by the map  
141 \* @throws IllegalStateException if <code>next()</code> has not yet been called  
142 \* @throws IllegalStateException if <code>remove()</code> has been called since the  
143 \* last call to <code>next()</code>  
144 \*/  
145 @Override  
146 public V setValue(final V value) {  
147 if (last == null) {  
148 throw new IllegalStateException("Iterator setValue() can only be called after next() and before remove()");  
149 }  
150 return last.setValue(value);  
151 }  
152  
153 //-----------------------------------------------------------------------  
154 /\*\*  
155 \* Resets the state of the iterator.  
156 \*/  
157 @Override  
158 public void reset() {  
159 iterator = map.entrySet().iterator();  
160 last = null;  
161 canRemove = false;  
162 }  
163  
164 /\*\*  
165 \* Gets the iterator as a String.  
166 \*  
167 \* @return a string version of the iterator  
168 \*/  
169 @Override  
170 public String toString() {  
171 if (last != null) {  
172 return "MapIterator[" + getKey() + "=" + getValue() + "]";  
173 }  
174 return "MapIterator[]";  
175 }  
176  
177}