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.util.Comparator;  
020import java.util.Iterator;  
021import java.util.ListIterator;  
022import java.util.Map;  
023import java.util.Set;  
024import java.util.SortedMap;  
025  
026import org.apache.commons.collections4.IterableSortedMap;  
027import org.apache.commons.collections4.OrderedMapIterator;  
028import org.apache.commons.collections4.iterators.ListIteratorWrapper;  
029  
030/\*\*  
031 \* Provides a base decorator that enables additional functionality to be added  
032 \* to a Map via decoration.  
033 \* <p>  
034 \* Methods are forwarded directly to the decorated map.  
035 \* </p>  
036 \* <p>  
037 \* This implementation does not perform any special processing with the map views.  
038 \* Instead it simply returns the set/collection from the wrapped map. This may be  
039 \* undesirable, for example if you are trying to write a validating implementation  
040 \* it would provide a loophole around the validation.  
041 \* But, you might want that loophole, so this class is kept simple.  
042 \* </p>  
043 \*  
044 \* @param <K> the type of the keys in the map  
045 \* @param <V> the type of the values in the map  
046 \* @since 3.0  
047 \*/  
048public abstract class AbstractSortedMapDecorator<K, V> extends AbstractMapDecorator<K, V> implements  
049 IterableSortedMap<K, V> {  
050  
051 /\*\*  
052 \* Constructor only used in deserialization, do not use otherwise.  
053 \* @since 3.1  
054 \*/  
055 protected AbstractSortedMapDecorator() {  
056 super();  
057 }  
058  
059 /\*\*  
060 \* Constructor that wraps (not copies).  
061 \*  
062 \* @param map the map to decorate, must not be null  
063 \* @throws NullPointerException if the map is null  
064 \*/  
065 public AbstractSortedMapDecorator(final SortedMap<K, V> map) {  
066 super(map);  
067 }  
068  
069 /\*\*  
070 \* Gets the map being decorated.  
071 \*  
072 \* @return the decorated map  
073 \*/  
074 @Override  
075 protected SortedMap<K, V> decorated() {  
076 return (SortedMap<K, V>) super.decorated();  
077 }  
078  
079 //-----------------------------------------------------------------------  
080 @Override  
081 public Comparator<? super K> comparator() {  
082 return decorated().comparator();  
083 }  
084  
085 @Override  
086 public K firstKey() {  
087 return decorated().firstKey();  
088 }  
089  
090 @Override  
091 public K lastKey() {  
092 return decorated().lastKey();  
093 }  
094  
095 @Override  
096 public SortedMap<K, V> subMap(final K fromKey, final K toKey) {  
097 return decorated().subMap(fromKey, toKey);  
098 }  
099  
100 @Override  
101 public SortedMap<K, V> headMap(final K toKey) {  
102 return decorated().headMap(toKey);  
103 }  
104  
105 @Override  
106 public SortedMap<K, V> tailMap(final K fromKey) {  
107 return decorated().tailMap(fromKey);  
108 }  
109  
110 @Override  
111 public K previousKey(final K key) {  
112 final SortedMap<K, V> headMap = headMap(key);  
113 return headMap.isEmpty() ? null : headMap.lastKey();  
114 }  
115  
116 @Override  
117 public K nextKey(final K key) {  
118 final Iterator<K> it = tailMap(key).keySet().iterator();  
119 it.next();  
120 return it.hasNext() ? it.next() : null;  
121 }  
122  
123 /\*\*  
124 \* {@inheritDoc}  
125 \*/  
126 @Override  
127 public OrderedMapIterator<K, V> mapIterator() {  
128 return new SortedMapIterator<>(entrySet());  
129 }  
130  
131 /\*\*  
132 \* OrderedMapIterator implementation.  
133 \*  
134 \* @param <K> the key type  
135 \* @param <V> the value type  
136 \*/  
137 protected static class SortedMapIterator<K, V> extends EntrySetToMapIteratorAdapter<K, V>  
138 implements OrderedMapIterator<K, V> {  
139  
140 /\*\*  
141 \* Create a new AbstractSortedMapDecorator.SortedMapIterator.  
142 \* @param entrySet the entrySet to iterate  
143 \*/  
144 protected SortedMapIterator(final Set<Map.Entry<K, V>> entrySet) {  
145 super(entrySet);  
146 }  
147  
148 /\*\*  
149 \* {@inheritDoc}  
150 \*/  
151 @Override  
152 public synchronized void reset() {  
153 super.reset();  
154 iterator = new ListIteratorWrapper<>(iterator);  
155 }  
156  
157 /\*\*  
158 \* {@inheritDoc}  
159 \*/  
160 @Override  
161 public boolean hasPrevious() {  
162 return ((ListIterator<Map.Entry<K, V>>) iterator).hasPrevious();  
163 }  
164  
165 /\*\*  
166 \* {@inheritDoc}  
167 \*/  
168 @Override  
169 public K previous() {  
170 entry = ((ListIterator<Map.Entry<K, V>>) iterator).previous();  
171 return getKey();  
172 }  
173 }  
174}