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.set;  
018  
019import java.io.Serializable;  
020import java.util.Collection;  
021import java.util.Iterator;  
022import java.util.Map;  
023import java.util.Set;  
024import java.util.function.Predicate;  
025  
026/\*\*  
027 \* Decorates a <code>Map</code> to obtain <code>Set</code> behaviour.  
028 \* <p>  
029 \* This class is used to create a <code>Set</code> with the same properties as  
030 \* the key set of any map. Thus, a ReferenceSet can be created by wrapping a  
031 \* <code>ReferenceMap</code> in an instance of this class.  
032 \* </p>  
033 \* <p>  
034 \* Most map implementation can be used to create a set by passing in dummy values.  
035 \* Exceptions include <code>BidiMap</code> implementations, as they require unique values.  
036 \* </p>  
037 \*  
038 \* @param <E> the type of the elements in this set  
039 \* @param <V> the dummy value type in this map  
040 \* @since 3.1  
041 \*/  
042public final class MapBackedSet<E, V> implements Set<E>, Serializable {  
043  
044 /\*\* Serialization version \*/  
045 private static final long serialVersionUID = 6723912213766056587L;  
046  
047 /\*\* The map being used as the backing store \*/  
048 private final Map<E, ? super V> map;  
049  
050 /\*\* The dummyValue to use \*/  
051 private final V dummyValue;  
052  
053 /\*\*  
054 \* Factory method to create a set from a map.  
055 \*  
056 \* @param <E> the element type  
057 \* @param <V> the dummy value type in the map  
058 \* @param map the map to decorate, must not be null  
059 \* @return a new map backed set  
060 \* @throws NullPointerException if map is null  
061 \* @since 4.0  
062 \*/  
063 public static <E, V> MapBackedSet<E, V> mapBackedSet(final Map<E, ? super V> map) {  
064 return mapBackedSet(map, null);  
065 }  
066  
067 /\*\*  
068 \* Factory method to create a set from a map.  
069 \*  
070 \* @param <E> the element type  
071 \* @param <V> the dummy value type in the map  
072 \* @param map the map to decorate, must not be null  
073 \* @param dummyValue the dummy value to use  
074 \* @return a new map backed set  
075 \* @throws NullPointerException if map is null  
076 \* @since 4.0  
077 \*/  
078 public static <E, V> MapBackedSet<E, V> mapBackedSet(final Map<E, ? super V> map, final V dummyValue) {  
079 return new MapBackedSet<>(map, dummyValue);  
080 }  
081  
082 //-----------------------------------------------------------------------  
083 /\*\*  
084 \* Constructor that wraps (not copies).  
085 \*  
086 \* @param map the map to decorate, must not be null  
087 \* @param dummyValue the dummy value to use  
088 \* @throws NullPointerException if map is null  
089 \*/  
090 private MapBackedSet(final Map<E, ? super V> map, final V dummyValue) {  
091 super();  
092 if (map == null) {  
093 throw new NullPointerException("The map must not be null");  
094 }  
095 this.map = map;  
096 this.dummyValue = dummyValue;  
097 }  
098  
099 //-----------------------------------------------------------------------  
100 @Override  
101 public int size() {  
102 return map.size();  
103 }  
104  
105 @Override  
106 public boolean isEmpty() {  
107 return map.isEmpty();  
108 }  
109  
110 @Override  
111 public Iterator<E> iterator() {  
112 return map.keySet().iterator();  
113 }  
114  
115 @Override  
116 public boolean contains(final Object obj) {  
117 return map.containsKey(obj);  
118 }  
119  
120 @Override  
121 public boolean containsAll(final Collection<?> coll) {  
122 return map.keySet().containsAll(coll);  
123 }  
124  
125 @Override  
126 public boolean add(final E obj) {  
127 final int size = map.size();  
128 map.put(obj, dummyValue);  
129 return map.size() != size;  
130 }  
131  
132 @Override  
133 public boolean addAll(final Collection<? extends E> coll) {  
134 final int size = map.size();  
135 for (final E e : coll) {  
136 map.put(e, dummyValue);  
137 }  
138 return map.size() != size;  
139 }  
140  
141 @Override  
142 public boolean remove(final Object obj) {  
143 final int size = map.size();  
144 map.remove(obj);  
145 return map.size() != size;  
146 }  
147  
148 /\*\*  
149 \* @since 4.4  
150 \*/  
151 @Override  
152 public boolean removeIf(Predicate<? super E> filter) {  
153 return map.keySet().removeIf(filter);  
154 }  
155  
156 @Override  
157 public boolean removeAll(final Collection<?> coll) {  
158 return map.keySet().removeAll(coll);  
159 }  
160  
161 @Override  
162 public boolean retainAll(final Collection<?> coll) {  
163 return map.keySet().retainAll(coll);  
164 }  
165  
166 @Override  
167 public void clear() {  
168 map.clear();  
169 }  
170  
171 @Override  
172 public Object[] toArray() {  
173 return map.keySet().toArray();  
174 }  
175  
176 @Override  
177 public <T> T[] toArray(final T[] array) {  
178 return map.keySet().toArray(array);  
179 }  
180  
181 @Override  
182 public boolean equals(final Object obj) {  
183 return map.keySet().equals(obj);  
184 }  
185  
186 @Override  
187 public int hashCode() {  
188 return map.keySet().hashCode();  
189 }  
190  
191}