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;  
018  
019import java.util.Set;  
020  
021/\*\*  
022 \* Defines a map that allows bidirectional lookup between key and values.  
023 \* <p>  
024 \* This extended <code>Map</code> represents a mapping where a key may  
025 \* lookup a value and a value may lookup a key with equal ease.  
026 \* This interface extends <code>Map</code> and so may be used anywhere a map  
027 \* is required. The interface provides an inverse map view, enabling  
028 \* full access to both directions of the <code>BidiMap</code>.  
029 \* </p>  
030 \* <p>  
031 \* Implementations should allow a value to be looked up from a key and  
032 \* a key to be looked up from a value with equal performance.  
033 \* </p>  
034 \* <p>  
035 \* This map enforces the restriction that there is a 1:1 relation between  
036 \* keys and values, meaning that multiple keys cannot map to the same value.  
037 \* This is required so that "inverting" the map results in a map without  
038 \* duplicate keys. See the {@link #put} method description for more information.  
039 \* </p>  
040 \*  
041 \* @param <K> the type of the keys in the map  
042 \* @param <V> the type of the values in the map  
043 \*  
044 \* @since 3.0  
045 \*/  
046public interface BidiMap<K, V> extends IterableMap<K, V> {  
047  
048 /\*\*  
049 \* Puts the key-value pair into the map, replacing any previous pair.  
050 \* <p>  
051 \* When adding a key-value pair, the value may already exist in the map  
052 \* against a different key. That mapping is removed, to ensure that the  
053 \* value only occurs once in the inverse map.  
054 \* </p>  
055 \* <pre>  
056 \* BidiMap map1 = new DualHashBidiMap();  
057 \* map.put("A","B"); // contains A mapped to B, as per Map  
058 \* map.put("A","C"); // contains A mapped to C, as per Map  
059 \*  
060 \* BidiMap map2 = new DualHashBidiMap();  
061 \* map.put("A","B"); // contains A mapped to B, as per Map  
062 \* map.put("C","B"); // contains C mapped to B, key A is removed  
063 \* </pre>  
064 \*  
065 \* @param key the key to store  
066 \* @param value the value to store  
067 \* @return the previous value mapped to this key  
068 \*  
069 \* @throws UnsupportedOperationException if the <code>put</code> method is not supported  
070 \* @throws ClassCastException (optional) if the map limits the type of the  
071 \* value and the specified value is inappropriate  
072 \* @throws IllegalArgumentException (optional) if the map limits the values  
073 \* in some way and the value was invalid  
074 \* @throws NullPointerException (optional) if the map limits the values to  
075 \* non-null and null was specified  
076 \*/  
077 @Override  
078 V put(K key, V value);  
079  
080 /\*\*  
081 \* Gets the key that is currently mapped to the specified value.  
082 \* <p>  
083 \* If the value is not contained in the map, <code>null</code> is returned.  
084 \* </p>  
085 \* <p>  
086 \* Implementations should seek to make this method perform equally as well  
087 \* as <code>get(Object)</code>.  
088 \* </p>  
089 \*  
090 \* @param value the value to find the key for  
091 \* @return the mapped key, or <code>null</code> if not found  
092 \*  
093 \* @throws ClassCastException (optional) if the map limits the type of the  
094 \* value and the specified value is inappropriate  
095 \* @throws NullPointerException (optional) if the map limits the values to  
096 \* non-null and null was specified  
097 \*/  
098 K getKey(Object value);  
099  
100 /\*\*  
101 \* Removes the key-value pair that is currently mapped to the specified  
102 \* value (optional operation).  
103 \* <p>  
104 \* If the value is not contained in the map, <code>null</code> is returned.  
105 \* </p>  
106 \* <p>  
107 \* Implementations should seek to make this method perform equally as well  
108 \* as <code>remove(Object)</code>.  
109 \* </p>  
110 \*  
111 \* @param value the value to find the key-value pair for  
112 \* @return the key that was removed, <code>null</code> if nothing removed  
113 \*  
114 \* @throws ClassCastException (optional) if the map limits the type of the  
115 \* value and the specified value is inappropriate  
116 \* @throws NullPointerException (optional) if the map limits the values to  
117 \* non-null and null was specified  
118 \* @throws UnsupportedOperationException if this method is not supported  
119 \* by the implementation  
120 \*/  
121 K removeValue(Object value);  
122  
123 /\*\*  
124 \* Gets a view of this map where the keys and values are reversed.  
125 \* <p>  
126 \* Changes to one map will be visible in the other and vice versa.  
127 \* This enables both directions of the map to be accessed as a <code>Map</code>.  
128 \* </p>  
129 \* <p>  
130 \* Implementations should seek to avoid creating a new object every time this  
131 \* method is called. See <code>AbstractMap.values()</code> etc. Calling this  
132 \* method on the inverse map should return the original.  
133 \* </p>  
134 \*  
135 \* @return an inverted bidirectional map  
136 \*/  
137 BidiMap<V, K> inverseBidiMap();  
138  
139 /\*\*  
140 \* Returns a {@link Set} view of the values contained in this map.  
141 \* The set is backed by the map, so changes to the map are reflected  
142 \* in the set, and vice-versa. If the map is modified while an iteration  
143 \* over the set is in progress (except through the iterator's own  
144 \* {@code remove} operation), the results of the iteration are undefined.  
145 \* The set supports element removal, which removes the corresponding  
146 \* mapping from the map, via the {@code Iterator.remove},  
147 \* {@code Collection.remove}, {@code removeAll},  
148 \* {@code retainAll} and {@code clear} operations. It does not  
149 \* support the {@code add} or {@code addAll} operations.  
150 \*  
151 \* @return a set view of the values contained in this map  
152 \*/  
153 @Override  
154 Set<V> values();  
155}