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.Collection;  
020  
021/\*\*  
022 \* Defines a map that holds a collection of values against each key.  
023 \* <p>  
024 \* A <code>MultiMap</code> is a Map with slightly different semantics.  
025 \* Putting a value into the map will add the value to a Collection at that key.  
026 \* Getting a value will return a Collection, holding all the values put to that key.  
027 \* </p>  
028 \* <p>  
029 \* For example:  
030 \* </p>  
031 \* <pre>  
032 \* MultiMap mhm = new MultiValueMap();  
033 \* mhm.put(key, "A");  
034 \* mhm.put(key, "B");  
035 \* mhm.put(key, "C");  
036 \* Collection coll = (Collection) mhm.get(key);</pre>  
037 \* <p>  
038 \* <code>coll</code> will be a collection containing "A", "B", "C".  
039 \* </p>  
040 \* <p>  
041 \* NOTE: Additional methods were added to this interface in Commons Collections 3.1.  
042 \* These were added solely for documentation purposes and do not change the interface  
043 \* as they were defined in the superinterface <code>Map</code> anyway.  
044 \* </p>  
045 \*  
046 \* @param <K> the type of the keys in this map  
047 \* @param <V> the type of the values in this map  
048 \*  
049 \* @since 2.0  
050 \* @deprecated since 4.1, use {@link MultiValuedMap} instead  
051 \*/  
052@Deprecated  
053public interface MultiMap<K, V> extends IterableMap<K, Object> {  
054  
055 /\*\*  
056 \* Removes a specific value from map.  
057 \* <p>  
058 \* The item is removed from the collection mapped to the specified key.  
059 \* Other values attached to that key are unaffected.  
060 \* <p>  
061 \* If the last value for a key is removed, implementations typically  
062 \* return <code>null</code> from a subsequent <code>get(Object)</code>, however  
063 \* they may choose to return an empty collection.  
064 \*  
065 \* @param key the key to remove from  
066 \* @param item the item to remove  
067 \* @return {@code true} if the mapping was removed, {@code false} otherwise  
068 \* @throws UnsupportedOperationException if the map is unmodifiable  
069 \* @throws ClassCastException if the key or value is of an invalid type  
070 \* @throws NullPointerException if the key or value is null and null is invalid  
071 \* @since 4.0 (signature in previous releases: V remove(K, V))  
072 \*/  
073 boolean removeMapping(K key, V item);  
074  
075 //-----------------------------------------------------------------------  
076 /\*\*  
077 \* Gets the number of keys in this map.  
078 \* <p>  
079 \* Implementations typically return only the count of keys in the map  
080 \* This cannot be mandated due to backwards compatibility of this interface.  
081 \*  
082 \* @return the number of key-collection mappings in this map  
083 \*/  
084 @Override  
085 int size();  
086  
087 /\*\*  
088 \* Gets the collection of values associated with the specified key.  
089 \* <p>  
090 \* The returned value will implement <code>Collection</code>. Implementations  
091 \* are free to declare that they return <code>Collection</code> subclasses  
092 \* such as <code>List</code> or <code>Set</code>.  
093 \* <p>  
094 \* Implementations typically return <code>null</code> if no values have  
095 \* been mapped to the key, however the implementation may choose to  
096 \* return an empty collection.  
097 \* <p>  
098 \* Implementations may choose to return a clone of the internal collection.  
099 \*  
100 \* @param key the key to retrieve  
101 \* @return the <code>Collection</code> of values, implementations should  
102 \* return <code>null</code> for no mapping, but may return an empty collection  
103 \* @throws ClassCastException if the key is of an invalid type  
104 \* @throws NullPointerException if the key is null and null keys are invalid  
105 \*/  
106 @Override  
107 Object get(Object key); // Cannot use get(K key) as that does not properly implement Map#get  
108  
109 /\*\*  
110 \* Checks whether the map contains the value specified.  
111 \* <p>  
112 \* Implementations typically check all collections against all keys for the value.  
113 \* This cannot be mandated due to backwards compatibility of this interface.  
114 \*  
115 \* @param value the value to search for  
116 \* @return true if the map contains the value  
117 \* @throws ClassCastException if the value is of an invalid type  
118 \* @throws NullPointerException if the value is null and null value are invalid  
119 \*/  
120 @Override  
121 boolean containsValue(Object value);  
122  
123 /\*\*  
124 \* Adds the value to the collection associated with the specified key.  
125 \* <p>  
126 \* Unlike a normal <code>Map</code> the previous value is not replaced.  
127 \* Instead the new value is added to the collection stored against the key.  
128 \* The collection may be a <code>List</code>, <code>Set</code> or other  
129 \* collection dependent on implementation.  
130 \*  
131 \* @param key the key to store against  
132 \* @param value the value to add to the collection at the key  
133 \* @return typically the value added if the map changed and null if the map did not change  
134 \* @throws UnsupportedOperationException if the map is unmodifiable  
135 \* @throws ClassCastException if the key or value is of an invalid type  
136 \* @throws NullPointerException if the key or value is null and null is invalid  
137 \* @throws IllegalArgumentException if the key or value is invalid  
138 \*/  
139 @Override  
140 Object put(K key, Object value);  
141  
142 /\*\*  
143 \* Removes all values associated with the specified key.  
144 \* <p>  
145 \* Implementations typically return <code>null</code> from a subsequent  
146 \* <code>get(Object)</code>, however they may choose to return an empty collection.  
147 \*  
148 \* @param key the key to remove values from  
149 \* @return the <code>Collection</code> of values removed, implementations should  
150 \* return <code>null</code> for no mapping found, but may return an empty collection  
151 \* @throws UnsupportedOperationException if the map is unmodifiable  
152 \* @throws ClassCastException if the key is of an invalid type  
153 \* @throws NullPointerException if the key is null and null keys are invalid  
154 \*/  
155 @Override  
156 Object remove(Object key); // Cannot use remove(K key) as that does not properly implement Map#remove  
157  
158 /\*\*  
159 \* Gets a collection containing all the values in the map.  
160 \* <p>  
161 \* Implementations typically return a collection containing the combination  
162 \* of values from all keys.  
163 \* This cannot be mandated due to backwards compatibility of this interface.  
164 \*  
165 \* @return a collection view of the values contained in this map  
166 \*/  
167 @Override  
168 Collection<Object> values();  
169  
170}