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.io.IOException;  
020import java.io.ObjectInputStream;  
021import java.io.ObjectOutputStream;  
022import java.io.Serializable;  
023  
024/\*\*  
025 \* A <code>Map</code> implementation that allows mappings to be  
026 \* removed by the garbage collector.  
027 \* <p>  
028 \* When you construct a <code>ReferenceMap</code>, you can specify what kind  
029 \* of references are used to store the map's keys and values.  
030 \* If non-hard references are used, then the garbage collector can remove  
031 \* mappings if a key or value becomes unreachable, or if the JVM's memory is  
032 \* running low. For information on how the different reference types behave,  
033 \* see {@link java.lang.ref.Reference Reference}.  
034 \* </p>  
035 \* <p>  
036 \* Different types of references can be specified for keys and values.  
037 \* The keys can be configured to be weak but the values hard,  
038 \* in which case this class will behave like a  
039 \* <a href="http://java.sun.com/j2se/1.4/docs/api/java/util/WeakHashMap.html">  
040 \* <code>WeakHashMap</code></a>. However, you can also specify hard keys and  
041 \* weak values, or any other combination. The default constructor uses  
042 \* hard keys and soft values, providing a memory-sensitive cache.  
043 \* </p>  
044 \* <p>  
045 \* This map is similar to  
046 \* {@link org.apache.commons.collections4.map.ReferenceIdentityMap ReferenceIdentityMap}.  
047 \* It differs in that keys and values in this class are compared using <code>equals()</code>.  
048 \* </p>  
049 \* <p>  
050 \* This {@link java.util.Map Map} implementation does <i>not</i> allow null elements.  
051 \* Attempting to add a null key or value to the map will raise a <code>NullPointerException</code>.  
052 \* </p>  
053 \* <p>  
054 \* This implementation is not synchronized.  
055 \* You can use {@link java.util.Collections#synchronizedMap} to  
056 \* provide synchronized access to a <code>ReferenceMap</code>.  
057 \* Remember that synchronization will not stop the garbage collector removing entries.  
058 \* </p>  
059 \* <p>  
060 \* All the available iterators can be reset back to the start by casting to  
061 \* <code>ResettableIterator</code> and calling <code>reset()</code>.  
062 \* </p>  
063 \* <p>  
064 \* <strong>Note that ReferenceMap is not synchronized and is not thread-safe.</strong>  
065 \* If you wish to use this map from multiple threads concurrently, you must use  
066 \* appropriate synchronization. The simplest approach is to wrap this map  
067 \* using {@link java.util.Collections#synchronizedMap}. This class may throw  
068 \* exceptions when accessed by concurrent threads without synchronization.  
069 \* </p>  
070 \* <p>  
071 \* NOTE: As from Commons Collections 3.1 this map extends <code>AbstractReferenceMap</code>  
072 \* (previously it extended AbstractMap). As a result, the implementation is now  
073 \* extensible and provides a <code>MapIterator</code>.  
074 \* </p>  
075 \*  
076 \* @param <K> the type of the keys in the map  
077 \* @param <V> the type of the values in the map  
078 \*  
079 \* @see java.lang.ref.Reference  
080 \* @since 3.0 (previously in main package v2.1)  
081 \*/  
082public class ReferenceMap<K, V> extends AbstractReferenceMap<K, V> implements Serializable {  
083  
084 /\*\* Serialization version \*/  
085 private static final long serialVersionUID = 1555089888138299607L;  
086  
087 /\*\*  
088 \* Constructs a new <code>ReferenceMap</code> that will  
089 \* use hard references to keys and soft references to values.  
090 \*/  
091 public ReferenceMap() {  
092 super(ReferenceStrength.HARD, ReferenceStrength.SOFT, DEFAULT\_CAPACITY,  
093 DEFAULT\_LOAD\_FACTOR, false);  
094 }  
095  
096 /\*\*  
097 \* Constructs a new <code>ReferenceMap</code> that will  
098 \* use the specified types of references.  
099 \*  
100 \* @param keyType the type of reference to use for keys;  
101 \* must be {@link AbstractReferenceMap.ReferenceStrength#HARD HARD},  
102 \* {@link AbstractReferenceMap.ReferenceStrength#SOFT SOFT},  
103 \* {@link AbstractReferenceMap.ReferenceStrength#WEAK WEAK}  
104 \* @param valueType the type of reference to use for values;  
105 \* must be {@link AbstractReferenceMap.ReferenceStrength#HARD HARD},  
106 \* {@link AbstractReferenceMap.ReferenceStrength#SOFT SOFT},  
107 \* {@link AbstractReferenceMap.ReferenceStrength#WEAK WEAK}  
108 \*/  
109 public ReferenceMap(final ReferenceStrength keyType, final ReferenceStrength valueType) {  
110 super(keyType, valueType, DEFAULT\_CAPACITY, DEFAULT\_LOAD\_FACTOR, false);  
111 }  
112  
113 /\*\*  
114 \* Constructs a new <code>ReferenceMap</code> that will  
115 \* use the specified types of references.  
116 \*  
117 \* @param keyType the type of reference to use for keys;  
118 \* must be {@link AbstractReferenceMap.ReferenceStrength#HARD HARD},  
119 \* {@link AbstractReferenceMap.ReferenceStrength#SOFT SOFT},  
120 \* {@link AbstractReferenceMap.ReferenceStrength#WEAK WEAK}  
121 \* @param valueType the type of reference to use for values;  
122 \* must be {@link AbstractReferenceMap.ReferenceStrength#HARD HARD},  
123 \* {@link AbstractReferenceMap.ReferenceStrength#SOFT SOFT},  
124 \* {@link AbstractReferenceMap.ReferenceStrength#WEAK WEAK}  
125 \* @param purgeValues should the value be automatically purged when the  
126 \* key is garbage collected  
127 \*/  
128 public ReferenceMap(final ReferenceStrength keyType, final ReferenceStrength valueType, final boolean purgeValues) {  
129 super(keyType, valueType, DEFAULT\_CAPACITY, DEFAULT\_LOAD\_FACTOR, purgeValues);  
130 }  
131  
132 /\*\*  
133 \* Constructs a new <code>ReferenceMap</code> with the  
134 \* specified reference types, load factor and initial  
135 \* capacity.  
136 \*  
137 \* @param keyType the type of reference to use for keys;  
138 \* must be {@link AbstractReferenceMap.ReferenceStrength#HARD HARD},  
139 \* {@link AbstractReferenceMap.ReferenceStrength#SOFT SOFT},  
140 \* {@link AbstractReferenceMap.ReferenceStrength#WEAK WEAK}  
141 \* @param valueType the type of reference to use for values;  
142 \* must be {@link AbstractReferenceMap.ReferenceStrength#HARD HARD},  
143 \* {@link AbstractReferenceMap.ReferenceStrength#SOFT SOFT},  
144 \* {@link AbstractReferenceMap.ReferenceStrength#WEAK WEAK}  
145 \* @param capacity the initial capacity for the map  
146 \* @param loadFactor the load factor for the map  
147 \*/  
148 public ReferenceMap(final ReferenceStrength keyType, final ReferenceStrength valueType, final int capacity,  
149 final float loadFactor) {  
150 super(keyType, valueType, capacity, loadFactor, false);  
151 }  
152  
153 /\*\*  
154 \* Constructs a new <code>ReferenceMap</code> with the  
155 \* specified reference types, load factor and initial  
156 \* capacity.  
157 \*  
158 \* @param keyType the type of reference to use for keys;  
159 \* must be {@link AbstractReferenceMap.ReferenceStrength#HARD HARD},  
160 \* {@link AbstractReferenceMap.ReferenceStrength#SOFT SOFT},  
161 \* {@link AbstractReferenceMap.ReferenceStrength#WEAK WEAK}  
162 \* @param valueType the type of reference to use for values;  
163 \* must be {@link AbstractReferenceMap.ReferenceStrength#HARD HARD},  
164 \* {@link AbstractReferenceMap.ReferenceStrength#SOFT SOFT},  
165 \* {@link AbstractReferenceMap.ReferenceStrength#WEAK WEAK}  
166 \* @param capacity the initial capacity for the map  
167 \* @param loadFactor the load factor for the map  
168 \* @param purgeValues should the value be automatically purged when the  
169 \* key is garbage collected  
170 \*/  
171 public ReferenceMap(final ReferenceStrength keyType, final ReferenceStrength valueType, final int capacity,  
172 final float loadFactor, final boolean purgeValues) {  
173 super(keyType, valueType, capacity, loadFactor, purgeValues);  
174 }  
175  
176 //-----------------------------------------------------------------------  
177 /\*\*  
178 \* Write the map out using a custom routine.  
179 \*  
180 \* @param out the output stream  
181 \* @throws IOException if an error occurs while writing to the stream  
182 \*/  
183 private void writeObject(final ObjectOutputStream out) throws IOException {  
184 out.defaultWriteObject();  
185 doWriteObject(out);  
186 }  
187  
188 /\*\*  
189 \* Read the map in using a custom routine.  
190 \*  
191 \* @param in the input stream  
192 \* @throws IOException if an error occurs while reading from the stream  
193 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
194 \*/  
195 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
196 in.defaultReadObject();  
197 doReadObject(in);  
198 }  
199  
200}