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;  
023import java.lang.ref.Reference;  
024  
025/\*\*  
026 \* A <code>Map</code> implementation that allows mappings to be  
027 \* removed by the garbage collector and matches keys and values based  
028 \* on <code>==</code> not <code>equals()</code>.  
029 \* <p>  
030 \* When you construct a <code>ReferenceIdentityMap</code>, you can specify what kind  
031 \* of references are used to store the map's keys and values.  
032 \* If non-hard references are used, then the garbage collector can remove  
033 \* mappings if a key or value becomes unreachable, or if the JVM's memory is  
034 \* running low. For information on how the different reference types behave,  
035 \* see {@link Reference}.  
036 \* </p>  
037 \* <p>  
038 \* Different types of references can be specified for keys and values.  
039 \* The default constructor uses hard keys and soft values, providing a  
040 \* memory-sensitive cache.  
041 \* </p>  
042 \* <p>  
043 \* This map is similar to  
044 \* {@link org.apache.commons.collections4.map.ReferenceMap ReferenceMap}.  
045 \* It differs in that keys and values in this class are compared using <code>==</code>.  
046 \* </p>  
047 \* <p>  
048 \* This map will violate the detail of various Map and map view contracts.  
049 \* As a general rule, don't compare this map to other maps.  
050 \* </p>  
051 \* <p>  
052 \* This {@link java.util.Map Map} implementation does <i>not</i> allow null elements.  
053 \* Attempting to add a null key or value to the map will raise a <code>NullPointerException</code>.  
054 \* </p>  
055 \* <p>  
056 \* This implementation is not synchronized.  
057 \* You can use {@link java.util.Collections#synchronizedMap} to  
058 \* provide synchronized access to a <code>ReferenceIdentityMap</code>.  
059 \* Remember that synchronization will not stop the garbage collector removing entries.  
060 \* </p>  
061 \* <p>  
062 \* All the available iterators can be reset back to the start by casting to  
063 \* <code>ResettableIterator</code> and calling <code>reset()</code>.  
064 \* </p>  
065 \* <p>  
066 \* <strong>Note that ReferenceIdentityMap is not synchronized and is not thread-safe.</strong>  
067 \* If you wish to use this map from multiple threads concurrently, you must use  
068 \* appropriate synchronization. The simplest approach is to wrap this map  
069 \* using {@link java.util.Collections#synchronizedMap}. This class may throw  
070 \* exceptions when accessed by concurrent threads without synchronization.  
071 \* </p>  
072 \*  
073 \* @param <K> the type of the keys in this map  
074 \* @param <V> the type of the values in this map  
075 \*  
076 \* @see java.lang.ref.Reference  
077 \* @since 3.0 (previously in main package v2.1)  
078 \*/  
079public class ReferenceIdentityMap<K, V> extends AbstractReferenceMap<K, V> implements Serializable {  
080  
081 /\*\* Serialization version \*/  
082 private static final long serialVersionUID = -1266190134568365852L;  
083  
084 /\*\*  
085 \* Constructs a new <code>ReferenceIdentityMap</code> that will  
086 \* use hard references to keys and soft references to values.  
087 \*/  
088 public ReferenceIdentityMap() {  
089 super(ReferenceStrength.HARD, ReferenceStrength.SOFT, DEFAULT\_CAPACITY,  
090 DEFAULT\_LOAD\_FACTOR, false);  
091 }  
092  
093 /\*\*  
094 \* Constructs a new <code>ReferenceIdentityMap</code> that will  
095 \* use the specified types of references.  
096 \*  
097 \* @param keyType the type of reference to use for keys;  
098 \* must be {@link AbstractReferenceMap.ReferenceStrength#HARD HARD},  
099 \* {@link AbstractReferenceMap.ReferenceStrength#SOFT SOFT},  
100 \* {@link AbstractReferenceMap.ReferenceStrength#WEAK WEAK}  
101 \* @param valueType the type of reference to use for values;  
102 \* must be {@link AbstractReferenceMap.ReferenceStrength#HARD HARD},  
103 \* {@link AbstractReferenceMap.ReferenceStrength#SOFT SOFT},  
104 \* {@link AbstractReferenceMap.ReferenceStrength#WEAK WEAK}  
105 \*/  
106 public ReferenceIdentityMap(final ReferenceStrength keyType, final ReferenceStrength valueType) {  
107 super(keyType, valueType, DEFAULT\_CAPACITY, DEFAULT\_LOAD\_FACTOR, false);  
108 }  
109  
110 /\*\*  
111 \* Constructs a new <code>ReferenceIdentityMap</code> that will  
112 \* use the specified types of references.  
113 \*  
114 \* @param keyType the type of reference to use for keys;  
115 \* must be {@link AbstractReferenceMap.ReferenceStrength#HARD HARD},  
116 \* {@link AbstractReferenceMap.ReferenceStrength#SOFT SOFT},  
117 \* {@link AbstractReferenceMap.ReferenceStrength#WEAK WEAK}  
118 \* @param valueType the type of reference to use for values;  
119 \* must be {@link AbstractReferenceMap.ReferenceStrength#HARD HARD},  
120 \* {@link AbstractReferenceMap.ReferenceStrength#SOFT SOFT},  
121 \* {@link AbstractReferenceMap.ReferenceStrength#WEAK WEAK}  
122 \* @param purgeValues should the value be automatically purged when the  
123 \* key is garbage collected  
124 \*/  
125 public ReferenceIdentityMap(final ReferenceStrength keyType, final ReferenceStrength valueType,  
126 final boolean purgeValues) {  
127 super(keyType, valueType, DEFAULT\_CAPACITY, DEFAULT\_LOAD\_FACTOR, purgeValues);  
128 }  
129  
130 /\*\*  
131 \* Constructs a new <code>ReferenceIdentityMap</code> with the  
132 \* specified reference types, load factor and initial capacity.  
133 \*  
134 \* @param keyType the type of reference to use for keys;  
135 \* must be {@link AbstractReferenceMap.ReferenceStrength#HARD HARD},  
136 \* {@link AbstractReferenceMap.ReferenceStrength#SOFT SOFT},  
137 \* {@link AbstractReferenceMap.ReferenceStrength#WEAK WEAK}  
138 \* @param valueType the type of reference to use for values;  
139 \* must be {@link AbstractReferenceMap.ReferenceStrength#HARD HARD},  
140 \* {@link AbstractReferenceMap.ReferenceStrength#SOFT SOFT},  
141 \* {@link AbstractReferenceMap.ReferenceStrength#WEAK WEAK}  
142 \* @param capacity the initial capacity for the map  
143 \* @param loadFactor the load factor for the map  
144 \*/  
145 public ReferenceIdentityMap(final ReferenceStrength keyType, final ReferenceStrength valueType,  
146 final int capacity, final float loadFactor) {  
147 super(keyType, valueType, capacity, loadFactor, false);  
148 }  
149  
150 /\*\*  
151 \* Constructs a new <code>ReferenceIdentityMap</code> with the  
152 \* specified reference types, load factor and initial capacity.  
153 \*  
154 \* @param keyType the type of reference to use for keys;  
155 \* must be {@link AbstractReferenceMap.ReferenceStrength#HARD HARD},  
156 \* {@link AbstractReferenceMap.ReferenceStrength#SOFT SOFT},  
157 \* {@link AbstractReferenceMap.ReferenceStrength#WEAK WEAK}  
158 \* @param valueType the type of reference to use for values;  
159 \* must be {@link AbstractReferenceMap.ReferenceStrength#HARD HARD},  
160 \* {@link AbstractReferenceMap.ReferenceStrength#SOFT SOFT},  
161 \* {@link AbstractReferenceMap.ReferenceStrength#WEAK WEAK}  
162 \* @param capacity the initial capacity for the map  
163 \* @param loadFactor the load factor for the map  
164 \* @param purgeValues should the value be automatically purged when the  
165 \* key is garbage collected  
166 \*/  
167 public ReferenceIdentityMap(final ReferenceStrength keyType, final ReferenceStrength valueType,  
168 final int capacity, final float loadFactor, final boolean purgeValues) {  
169 super(keyType, valueType, capacity, loadFactor, purgeValues);  
170 }  
171  
172 //-----------------------------------------------------------------------  
173 /\*\*  
174 \* Gets the hash code for the key specified.  
175 \* <p>  
176 \* This implementation uses the identity hash code.  
177 \*  
178 \* @param key the key to get a hash code for  
179 \* @return the hash code  
180 \*/  
181 @Override  
182 protected int hash(final Object key) {  
183 return System.identityHashCode(key);  
184 }  
185  
186 /\*\*  
187 \* Gets the hash code for a MapEntry.  
188 \* <p>  
189 \* This implementation uses the identity hash code.  
190 \*  
191 \* @param key the key to get a hash code for, may be null  
192 \* @param value the value to get a hash code for, may be null  
193 \* @return the hash code, as per the MapEntry specification  
194 \*/  
195 @Override  
196 protected int hashEntry(final Object key, final Object value) {  
197 return System.identityHashCode(key) ^  
198 System.identityHashCode(value);  
199 }  
200  
201 /\*\*  
202 \* Compares two keys for equals.  
203 \* <p>  
204 \* This implementation converts the key from the entry to a real reference  
205 \* before comparison and uses <code>==</code>.  
206 \*  
207 \* @param key1 the first key to compare passed in from outside  
208 \* @param key2 the second key extracted from the entry via <code>entry.key</code>  
209 \* @return true if equal by identity  
210 \*/  
211 @Override  
212 protected boolean isEqualKey(final Object key1, Object key2) {  
213 key2 = isKeyType(ReferenceStrength.HARD) ? key2 : ((Reference<?>) key2).get();  
214 return key1 == key2;  
215 }  
216  
217 /\*\*  
218 \* Compares two values for equals.  
219 \* <p>  
220 \* This implementation uses <code>==</code>.  
221 \*  
222 \* @param value1 the first value to compare passed in from outside  
223 \* @param value2 the second value extracted from the entry via <code>getValue()</code>  
224 \* @return true if equal by identity  
225 \*/  
226 @Override  
227 protected boolean isEqualValue(final Object value1, final Object value2) {  
228 return value1 == value2;  
229 }  
230  
231 //-----------------------------------------------------------------------  
232 /\*\*  
233 \* Write the map out using a custom routine.  
234 \*  
235 \* @param out the output stream  
236 \* @throws IOException if an error occurs while writing to the stream  
237 \*/  
238 private void writeObject(final ObjectOutputStream out) throws IOException {  
239 out.defaultWriteObject();  
240 doWriteObject(out);  
241 }  
242  
243 /\*\*  
244 \* Read the map in using a custom routine.  
245 \*  
246 \* @param in the input stream  
247 \* @throws IOException if an error occurs while reading from the stream  
248 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
249 \*/  
250 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
251 in.defaultReadObject();  
252 doReadObject(in);  
253 }  
254  
255}