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.util.Map;  
024  
025/\*\*  
026 \* A case-insensitive <code>Map</code>.  
027 \* <p>  
028 \* Before keys are added to the map or compared to other existing keys, they are converted  
029 \* to all lowercase in a locale-independent fashion by using information from the Unicode  
030 \* data file.  
031 \* </p>  
032 \* <p>  
033 \* Null keys are supported.  
034 \* </p>  
035 \* <p>  
036 \* The <code>keySet()</code> method returns all lowercase keys, or nulls.  
037 \* </p>  
038 \* <p>  
039 \* Example:  
040 \* </p>  
041 \* <pre><code>  
042 \* Map<String, String> map = new CaseInsensitiveMap<String, String>();  
043 \* map.put("One", "One");  
044 \* map.put("Two", "Two");  
045 \* map.put(null, "Three");  
046 \* map.put("one", "Four");  
047 \* </code></pre>  
048 \* <p>  
049 \* The example above creates a <code>CaseInsensitiveMap</code> with three entries.  
050 \* </p>  
051 \* <p>  
052 \* <code>map.get(null)</code> returns <code>"Three"</code> and <code>map.get("ONE")</code>  
053 \* returns <code>"Four".</code> The <code>Set</code> returned by <code>keySet()</code>  
054 \* equals <code>{"one", "two", null}.</code>  
055 \* </p>  
056 \* <p>  
057 \* <strong>This map will violate the detail of various Map and map view contracts.</strong>  
058 \* As a general rule, don't compare this map to other maps. In particular, you can't  
059 \* use decorators like {@link ListOrderedMap} on it, which silently assume that these  
060 \* contracts are fulfilled.  
061 \* </p>  
062 \* <p>  
063 \* <strong>Note that CaseInsensitiveMap is not synchronized and is not thread-safe.</strong>  
064 \* If you wish to use this map from multiple threads concurrently, you must use  
065 \* appropriate synchronization. The simplest approach is to wrap this map  
066 \* using {@link java.util.Collections#synchronizedMap(Map)}. This class may throw  
067 \* exceptions when accessed by concurrent threads without synchronization.  
068 \* </p>  
069 \*  
070 \* @param <K> the type of the keys in this map  
071 \* @param <V> the type of the values in this map  
072 \* @since 3.0  
073 \*/  
074public class CaseInsensitiveMap<K, V> extends AbstractHashedMap<K, V> implements Serializable, Cloneable {  
075  
076 /\*\* Serialisation version \*/  
077 private static final long serialVersionUID = -7074655917369299456L;  
078  
079 /\*\*  
080 \* Constructs a new empty map with default size and load factor.  
081 \*/  
082 public CaseInsensitiveMap() {  
083 super(DEFAULT\_CAPACITY, DEFAULT\_LOAD\_FACTOR, DEFAULT\_THRESHOLD);  
084 }  
085  
086 /\*\*  
087 \* Constructs a new, empty map with the specified initial capacity.  
088 \*  
089 \* @param initialCapacity the initial capacity  
090 \* @throws IllegalArgumentException if the initial capacity is negative  
091 \*/  
092 public CaseInsensitiveMap(final int initialCapacity) {  
093 super(initialCapacity);  
094 }  
095  
096 /\*\*  
097 \* Constructs a new, empty map with the specified initial capacity and  
098 \* load factor.  
099 \*  
100 \* @param initialCapacity the initial capacity  
101 \* @param loadFactor the load factor  
102 \* @throws IllegalArgumentException if the initial capacity is negative  
103 \* @throws IllegalArgumentException if the load factor is less than zero  
104 \*/  
105 public CaseInsensitiveMap(final int initialCapacity, final float loadFactor) {  
106 super(initialCapacity, loadFactor);  
107 }  
108  
109 /\*\*  
110 \* Constructor copying elements from another map.  
111 \* <p>  
112 \* Keys will be converted to lower case strings, which may cause  
113 \* some entries to be removed (if string representation of keys differ  
114 \* only by character case).  
115 \*  
116 \* @param map the map to copy  
117 \* @throws NullPointerException if the map is null  
118 \*/  
119 public CaseInsensitiveMap(final Map<? extends K, ? extends V> map) {  
120 super(map);  
121 }  
122  
123 //-----------------------------------------------------------------------  
124 /\*\*  
125 \* Overrides convertKey() from {@link AbstractHashedMap} to convert keys to  
126 \* lower case.  
127 \* <p>  
128 \* Returns {@link AbstractHashedMap#NULL} if key is null.  
129 \*  
130 \* @param key the key convert  
131 \* @return the converted key  
132 \*/  
133 @Override  
134 protected Object convertKey(final Object key) {  
135 if (key != null) {  
136 final char[] chars = key.toString().toCharArray();  
137 for (int i = chars.length - 1; i >= 0; i--) {  
138 chars[i] = Character.toLowerCase(Character.toUpperCase(chars[i]));  
139 }  
140 return new String(chars);  
141 }  
142 return AbstractHashedMap.NULL;  
143 }  
144  
145 //-----------------------------------------------------------------------  
146 /\*\*  
147 \* Clones the map without cloning the keys or values.  
148 \*  
149 \* @return a shallow clone  
150 \*/  
151 @Override  
152 public CaseInsensitiveMap<K, V> clone() {  
153 return (CaseInsensitiveMap<K, V>) super.clone();  
154 }  
155  
156 /\*\*  
157 \* Write the map out using a custom routine.  
158 \*  
159 \* @param out the output stream  
160 \* @throws IOException if an error occurs while writing to the stream  
161 \*/  
162 private void writeObject(final ObjectOutputStream out) throws IOException {  
163 out.defaultWriteObject();  
164 doWriteObject(out);  
165 }  
166  
167 /\*\*  
168 \* Read the map in using a custom routine.  
169 \*  
170 \* @param in the input stream  
171 \* @throws IOException if an error occurs while reading from the stream  
172 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
173 \*/  
174 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
175 in.defaultReadObject();  
176 doReadObject(in);  
177 }  
178  
179}