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 <code>Map</code> implementation that is a general purpose alternative  
027 \* to <code>HashMap</code>.  
028 \* <p>  
029 \* This implementation improves on the JDK1.4 HashMap by adding the  
030 \* {@link org.apache.commons.collections4.MapIterator MapIterator}  
031 \* functionality and many methods for subclassing.  
032 \* </p>  
033 \* <p>  
034 \* <strong>Note that HashedMap is not synchronized and is not thread-safe.</strong>  
035 \* If you wish to use this map from multiple threads concurrently, you must use  
036 \* appropriate synchronization. The simplest approach is to wrap this map  
037 \* using {@link java.util.Collections#synchronizedMap(Map)}. This class may throw  
038 \* exceptions when accessed by concurrent threads without synchronization.  
039 \* </p>  
040 \*  
041 \* @param <K> the type of the keys in this map  
042 \* @param <V> the type of the values in this map  
043 \* @since 3.0  
044 \*/  
045public class HashedMap<K, V>  
046 extends AbstractHashedMap<K, V> implements Serializable, Cloneable {  
047  
048 /\*\* Serialisation version \*/  
049 private static final long serialVersionUID = -1788199231038721040L;  
050  
051 /\*\*  
052 \* Constructs a new empty map with default size and load factor.  
053 \*/  
054 public HashedMap() {  
055 super(DEFAULT\_CAPACITY, DEFAULT\_LOAD\_FACTOR, DEFAULT\_THRESHOLD);  
056 }  
057  
058 /\*\*  
059 \* Constructs a new, empty map with the specified initial capacity.  
060 \*  
061 \* @param initialCapacity the initial capacity  
062 \* @throws IllegalArgumentException if the initial capacity is negative  
063 \*/  
064 public HashedMap(final int initialCapacity) {  
065 super(initialCapacity);  
066 }  
067  
068 /\*\*  
069 \* Constructs a new, empty map with the specified initial capacity and  
070 \* load factor.  
071 \*  
072 \* @param initialCapacity the initial capacity  
073 \* @param loadFactor the load factor  
074 \* @throws IllegalArgumentException if the initial capacity is negative  
075 \* @throws IllegalArgumentException if the load factor is less than zero  
076 \*/  
077 public HashedMap(final int initialCapacity, final float loadFactor) {  
078 super(initialCapacity, loadFactor);  
079 }  
080  
081 /\*\*  
082 \* Constructor copying elements from another map.  
083 \*  
084 \* @param map the map to copy  
085 \* @throws NullPointerException if the map is null  
086 \*/  
087 public HashedMap(final Map<? extends K, ? extends V> map) {  
088 super(map);  
089 }  
090  
091 //-----------------------------------------------------------------------  
092 /\*\*  
093 \* Clones the map without cloning the keys or values.  
094 \*  
095 \* @return a shallow clone  
096 \*/  
097 @Override  
098 public HashedMap<K, V> clone() {  
099 return (HashedMap<K, V>) super.clone();  
100 }  
101  
102 /\*\*  
103 \* Write the map out using a custom routine.  
104 \*  
105 \* @param out the output stream  
106 \* @throws IOException if an error occurs while writing to the stream  
107 \*/  
108 private void writeObject(final ObjectOutputStream out) throws IOException {  
109 out.defaultWriteObject();  
110 doWriteObject(out);  
111 }  
112  
113 /\*\*  
114 \* Read the map in using a custom routine.  
115 \*  
116 \* @param in the input stream  
117 \* @throws IOException if an error occurs while reading from the stream  
118 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
119 \*/  
120 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
121 in.defaultReadObject();  
122 doReadObject(in);  
123 }  
124  
125}