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.Collection;  
024import java.util.Map;  
025import java.util.Set;  
026  
027import org.apache.commons.collections4.BoundedMap;  
028import org.apache.commons.collections4.collection.UnmodifiableCollection;  
029import org.apache.commons.collections4.set.UnmodifiableSet;  
030  
031/\*\*  
032 \* Decorates another <code>Map</code> to fix the size, preventing add/remove.  
033 \* <p>  
034 \* Any action that would change the size of the map is disallowed.  
035 \* The put method is allowed to change the value associated with an existing  
036 \* key however.  
037 \* </p>  
038 \* <p>  
039 \* If trying to remove or clear the map, an UnsupportedOperationException is  
040 \* thrown. If trying to put a new mapping into the map, an  
041 \* IllegalArgumentException is thrown. This is because the put method can  
042 \* succeed if the mapping's key already exists in the map, so the put method  
043 \* is not always unsupported.  
044 \* </p>  
045 \* <p>  
046 \* <strong>Note that FixedSizeMap is not synchronized and is not thread-safe.</strong>  
047 \* If you wish to use this map from multiple threads concurrently, you must use  
048 \* appropriate synchronization. The simplest approach is to wrap this map  
049 \* using {@link java.util.Collections#synchronizedMap(Map)}. This class may throw  
050 \* exceptions when accessed by concurrent threads without synchronization.  
051 \* </p>  
052 \* <p>  
053 \* This class is Serializable from Commons Collections 3.1.  
054 \* </p>  
055 \*  
056 \* @param <K> the type of the keys in this map  
057 \* @param <V> the type of the values in this map  
058 \* @since 3.0  
059 \*/  
060public class FixedSizeMap<K, V>  
061 extends AbstractMapDecorator<K, V>  
062 implements BoundedMap<K, V>, Serializable {  
063  
064 /\*\* Serialization version \*/  
065 private static final long serialVersionUID = 7450927208116179316L;  
066  
067 /\*\*  
068 \* Factory method to create a fixed size map.  
069 \*  
070 \* @param <K> the key type  
071 \* @param <V> the value type  
072 \* @param map the map to decorate, must not be null  
073 \* @return a new fixed size map  
074 \* @throws NullPointerException if map is null  
075 \* @since 4.0  
076 \*/  
077 public static <K, V> FixedSizeMap<K, V> fixedSizeMap(final Map<K, V> map) {  
078 return new FixedSizeMap<>(map);  
079 }  
080  
081 //-----------------------------------------------------------------------  
082 /\*\*  
083 \* Constructor that wraps (not copies).  
084 \*  
085 \* @param map the map to decorate, must not be null  
086 \* @throws NullPointerException if map is null  
087 \*/  
088 protected FixedSizeMap(final Map<K, V> map) {  
089 super(map);  
090 }  
091  
092 //-----------------------------------------------------------------------  
093 /\*\*  
094 \* Write the map out using a custom routine.  
095 \*  
096 \* @param out the output stream  
097 \* @throws IOException if an error occurs while writing to the stream  
098 \* @since 3.1  
099 \*/  
100 private void writeObject(final ObjectOutputStream out) throws IOException {  
101 out.defaultWriteObject();  
102 out.writeObject(map);  
103 }  
104  
105 /\*\*  
106 \* Read the map in using a custom routine.  
107 \*  
108 \* @param in the input stream  
109 \* @throws IOException if an error occurs while reading from the stream  
110 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
111 \* @since 3.1  
112 \*/  
113 @SuppressWarnings("unchecked") // (1) should only fail if input stream is incorrect  
114 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
115 in.defaultReadObject();  
116 map = (Map<K, V>) in.readObject(); // (1)  
117 }  
118  
119 //-----------------------------------------------------------------------  
120 @Override  
121 public V put(final K key, final V value) {  
122 if (map.containsKey(key) == false) {  
123 throw new IllegalArgumentException("Cannot put new key/value pair - Map is fixed size");  
124 }  
125 return map.put(key, value);  
126 }  
127  
128 @Override  
129 public void putAll(final Map<? extends K, ? extends V> mapToCopy) {  
130 for (final K key : mapToCopy.keySet()) {  
131 if (!containsKey(key)) {  
132 throw new IllegalArgumentException("Cannot put new key/value pair - Map is fixed size");  
133 }  
134 }  
135 map.putAll(mapToCopy);  
136 }  
137  
138 @Override  
139 public void clear() {  
140 throw new UnsupportedOperationException("Map is fixed size");  
141 }  
142  
143 @Override  
144 public V remove(final Object key) {  
145 throw new UnsupportedOperationException("Map is fixed size");  
146 }  
147  
148 @Override  
149 public Set<Map.Entry<K, V>> entrySet() {  
150 final Set<Map.Entry<K, V>> set = map.entrySet();  
151 // unmodifiable set will still allow modification via Map.Entry objects  
152 return UnmodifiableSet.unmodifiableSet(set);  
153 }  
154  
155 @Override  
156 public Set<K> keySet() {  
157 final Set<K> set = map.keySet();  
158 return UnmodifiableSet.unmodifiableSet(set);  
159 }  
160  
161 @Override  
162 public Collection<V> values() {  
163 final Collection<V> coll = map.values();  
164 return UnmodifiableCollection.unmodifiableCollection(coll);  
165 }  
166  
167 @Override  
168 public boolean isFull() {  
169 return true;  
170 }  
171  
172 @Override  
173 public int maxSize() {  
174 return size();  
175 }  
176  
177}