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.HashMap;  
024import java.util.Map;  
025  
026import org.apache.commons.collections4.Factory;  
027import org.apache.commons.collections4.Transformer;  
028import org.apache.commons.collections4.functors.ConstantTransformer;  
029import org.apache.commons.collections4.functors.FactoryTransformer;  
030  
031/\*\*  
032 \* Decorates another <code>Map</code> returning a default value if the map  
033 \* does not contain the requested key.  
034 \* <p>  
035 \* When the {@link #get(Object)} method is called with a key that does not  
036 \* exist in the map, this map will return the default value specified in  
037 \* the constructor/factory. Only the get method is altered, so the  
038 \* {@link Map#containsKey(Object)} can be used to determine if a key really  
039 \* is in the map or not.  
040 \* </p>  
041 \* <p>  
042 \* The defaulted value is not added to the map.  
043 \* Compare this behaviour with {@link LazyMap}, which does add the value  
044 \* to the map (via a Transformer).  
045 \* </p>  
046 \* <p>  
047 \* For instance:  
048 \* </p>  
049 \* <pre>  
050 \* Map map = new DefaultedMap("NULL");  
051 \* Object obj = map.get("Surname");  
052 \* // obj == "NULL"  
053 \* </pre>  
054 \* <p>  
055 \* After the above code is executed the map is still empty.  
056 \* </p>  
057 \* <p>  
058 \* <strong>Note that DefaultedMap is not synchronized and is not thread-safe.</strong>  
059 \* If you wish to use this map from multiple threads concurrently, you must use  
060 \* appropriate synchronization. The simplest approach is to wrap this map  
061 \* using {@link java.util.Collections#synchronizedMap(Map)}. This class may throw  
062 \* exceptions when accessed by concurrent threads without synchronization.  
063 \* </p>  
064 \*  
065 \* @param <K> the type of the keys in this map  
066 \* @param <V> the type of the values in this map  
067 \*  
068 \* @since 3.2  
069 \* @see LazyMap  
070 \*/  
071public class DefaultedMap<K, V> extends AbstractMapDecorator<K, V> implements Serializable {  
072  
073 /\*\* Serialization version \*/  
074 private static final long serialVersionUID = 19698628745827L;  
075  
076 /\*\* The transformer to use if the map does not contain a key \*/  
077 private final Transformer<? super K, ? extends V> value;  
078  
079 //-----------------------------------------------------------------------  
080 /\*\*  
081 \* Factory method to create a defaulting map.  
082 \* <p>  
083 \* The value specified is returned when a missing key is found.  
084 \*  
085 \* @param <K> the key type  
086 \* @param <V> the value type  
087 \* @param map the map to decorate, must not be null  
088 \* @param defaultValue the default value to return when the key is not found  
089 \* @return a new defaulting map  
090 \* @throws NullPointerException if map is null  
091 \* @since 4.0  
092 \*/  
093 public static <K, V> DefaultedMap<K, V> defaultedMap(final Map<K, V> map, final V defaultValue) {  
094 return new DefaultedMap<>(map, ConstantTransformer.constantTransformer(defaultValue));  
095 }  
096  
097 /\*\*  
098 \* Factory method to create a defaulting map.  
099 \* <p>  
100 \* The factory specified is called when a missing key is found.  
101 \* The result will be returned as the result of the map get(key) method.  
102 \*  
103 \* @param <K> the key type  
104 \* @param <V> the value type  
105 \* @param map the map to decorate, must not be null  
106 \* @param factory the factory to use to create entries, must not be null  
107 \* @return a new defaulting map  
108 \* @throws NullPointerException if map or factory is null  
109 \* @since 4.0  
110 \*/  
111 public static <K, V> DefaultedMap<K, V> defaultedMap(final Map<K, V> map, final Factory<? extends V> factory) {  
112 if (factory == null) {  
113 throw new IllegalArgumentException("Factory must not be null");  
114 }  
115 return new DefaultedMap<>(map, FactoryTransformer.factoryTransformer(factory));  
116 }  
117  
118 /\*\*  
119 \* Factory method to create a defaulting map.  
120 \* <p>  
121 \* The transformer specified is called when a missing key is found.  
122 \* The key is passed to the transformer as the input, and the result  
123 \* will be returned as the result of the map get(key) method.  
124 \*  
125 \* @param <K> the key type  
126 \* @param <V> the value type  
127 \* @param map the map to decorate, must not be null  
128 \* @param transformer the transformer to use as a factory to create entries, must not be null  
129 \* @return a new defaulting map  
130 \* @throws NullPointerException if map or factory is null  
131 \* @since 4.0  
132 \*/  
133 public static <K, V> Map<K, V> defaultedMap(final Map<K, V> map,  
134 final Transformer<? super K, ? extends V> transformer) {  
135 if (transformer == null) {  
136 throw new IllegalArgumentException("Transformer must not be null");  
137 }  
138 return new DefaultedMap<>(map, transformer);  
139 }  
140  
141 //-----------------------------------------------------------------------  
142 /\*\*  
143 \* Constructs a new empty <code>DefaultedMap</code> that decorates  
144 \* a <code>HashMap</code>.  
145 \* <p>  
146 \* The object passed in will be returned by the map whenever an  
147 \* unknown key is requested.  
148 \*  
149 \* @param defaultValue the default value to return when the key is not found  
150 \*/  
151 public DefaultedMap(final V defaultValue) {  
152 this(ConstantTransformer.constantTransformer(defaultValue));  
153 }  
154  
155 /\*\*  
156 \* Constructs a new empty <code>DefaultedMap</code> that decorates a <code>HashMap</code>.  
157 \*  
158 \* @param defaultValueTransformer transformer to use to generate missing values.  
159 \*/  
160 public DefaultedMap(final Transformer<? super K, ? extends V> defaultValueTransformer) {  
161 this(new HashMap<K, V>(), defaultValueTransformer);  
162 }  
163  
164 /\*\*  
165 \* Constructor that wraps (not copies).  
166 \*  
167 \* @param map the map to decorate, must not be null  
168 \* @param defaultValueTransformer the value transformer to use  
169 \* @throws NullPointerException if map or transformer is null  
170 \*/  
171 protected DefaultedMap(final Map<K, V> map, final Transformer<? super K, ? extends V> defaultValueTransformer) {  
172 super(map);  
173 if (defaultValueTransformer == null) {  
174 throw new NullPointerException("Transformer must not be null.");  
175 }  
176 this.value = defaultValueTransformer;  
177 }  
178  
179 //-----------------------------------------------------------------------  
180 /\*\*  
181 \* Write the map out using a custom routine.  
182 \*  
183 \* @param out the output stream  
184 \* @throws IOException if an error occurs while writing to the stream  
185 \*/  
186 private void writeObject(final ObjectOutputStream out) throws IOException {  
187 out.defaultWriteObject();  
188 out.writeObject(map);  
189 }  
190  
191 /\*\*  
192 \* Read the map in using a custom routine.  
193 \*  
194 \* @param in the input stream  
195 \* @throws IOException if an error occurs while reading from the stream  
196 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
197 \*/  
198 @SuppressWarnings("unchecked")  
199 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
200 in.defaultReadObject();  
201 map = (Map<K, V>) in.readObject();  
202 }  
203  
204 //-----------------------------------------------------------------------  
205 @Override  
206 @SuppressWarnings("unchecked")  
207 public V get(final Object key) {  
208 V v;  
209 return (((v = map.get(key)) != null) || map.containsKey(key))  
210 ? v  
211 : value.transform((K) key);  
212 }  
213  
214 // no need to wrap keySet, entrySet or values as they are views of  
215 // existing map entries - you can't do a map-style get on them.  
216}