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.util.Collection;  
020import java.util.Map;  
021import java.util.Set;  
022  
023/\*\*  
024 \* Provides a base decorator that enables additional functionality to be added  
025 \* to a Map via decoration.  
026 \* <p>  
027 \* Methods are forwarded directly to the decorated map.  
028 \* </p>  
029 \* <p>  
030 \* This implementation does not perform any special processing with  
031 \* {@link #entrySet()}, {@link #keySet()} or {@link #values()}. Instead  
032 \* it simply returns the set/collection from the wrapped map. This may be  
033 \* undesirable, for example if you are trying to write a validating  
034 \* implementation it would provide a loophole around the validation.  
035 \* But, you might want that loophole, so this class is kept simple.  
036 \* </p>  
037 \*  
038 \* @param <K> the type of the keys in the map  
039 \* @param <V> the type of the values in the map  
040 \* @since 3.0  
041 \*/  
042public abstract class AbstractMapDecorator<K, V> extends AbstractIterableMap<K, V> {  
043  
044 /\*\* The map to decorate \*/  
045 transient Map<K, V> map;  
046  
047 /\*\*  
048 \* Constructor only used in deserialization, do not use otherwise.  
049 \* @since 3.1  
050 \*/  
051 protected AbstractMapDecorator() {  
052 super();  
053 }  
054  
055 /\*\*  
056 \* Constructor that wraps (not copies).  
057 \*  
058 \* @param map the map to decorate, must not be null  
059 \* @throws NullPointerException if the map is null  
060 \*/  
061 protected AbstractMapDecorator(final Map<K, V> map) {  
062 if (map == null) {  
063 throw new NullPointerException("Map must not be null.");  
064 }  
065 this.map = map;  
066 }  
067  
068 /\*\*  
069 \* Gets the map being decorated.  
070 \*  
071 \* @return the decorated map  
072 \*/  
073 protected Map<K, V> decorated() {  
074 return map;  
075 }  
076  
077 //-----------------------------------------------------------------------  
078 @Override  
079 public void clear() {  
080 decorated().clear();  
081 }  
082  
083 @Override  
084 public boolean containsKey(final Object key) {  
085 return decorated().containsKey(key);  
086 }  
087  
088 @Override  
089 public boolean containsValue(final Object value) {  
090 return decorated().containsValue(value);  
091 }  
092  
093 @Override  
094 public Set<Map.Entry<K, V>> entrySet() {  
095 return decorated().entrySet();  
096 }  
097  
098 @Override  
099 public V get(final Object key) {  
100 return decorated().get(key);  
101 }  
102  
103 @Override  
104 public boolean isEmpty() {  
105 return decorated().isEmpty();  
106 }  
107  
108 @Override  
109 public Set<K> keySet() {  
110 return decorated().keySet();  
111 }  
112  
113 @Override  
114 public V put(final K key, final V value) {  
115 return decorated().put(key, value);  
116 }  
117  
118 @Override  
119 public void putAll(final Map<? extends K, ? extends V> mapToCopy) {  
120 decorated().putAll(mapToCopy);  
121 }  
122  
123 @Override  
124 public V remove(final Object key) {  
125 return decorated().remove(key);  
126 }  
127  
128 @Override  
129 public int size() {  
130 return decorated().size();  
131 }  
132  
133 @Override  
134 public Collection<V> values() {  
135 return decorated().values();  
136 }  
137  
138 @Override  
139 public boolean equals(final Object object) {  
140 if (object == this) {  
141 return true;  
142 }  
143 return decorated().equals(object);  
144 }  
145  
146 @Override  
147 public int hashCode() {  
148 return decorated().hashCode();  
149 }  
150  
151 @Override  
152 public String toString() {  
153 return decorated().toString();  
154 }  
155  
156}