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.collection;  
018  
019import java.io.Serializable;  
020import java.util.Collection;  
021import java.util.Iterator;  
022import java.util.function.Predicate;  
023  
024/\*\*  
025 \* Decorates another <code>Collection</code> to provide additional behaviour.  
026 \* <p>  
027 \* Each method call made on this <code>Collection</code> is forwarded to the  
028 \* decorated <code>Collection</code>. This class is used as a framework on which  
029 \* to build to extensions such as synchronized and unmodifiable behaviour. The  
030 \* main advantage of decoration is that one decorator can wrap any implementation  
031 \* of <code>Collection</code>, whereas sub-classing requires a new class to be  
032 \* written for each implementation.  
033 \* </p>  
034 \* <p>  
035 \* This implementation does not perform any special processing with  
036 \* {@link #iterator()}. Instead it simply returns the value from the  
037 \* wrapped collection. This may be undesirable, for example if you are trying  
038 \* to write an unmodifiable implementation it might provide a loophole.  
039 \* </p>  
040 \* <p>  
041 \* This implementation does not forward the hashCode and equals methods through  
042 \* to the backing object, but relies on Object's implementation. This is necessary  
043 \* to preserve the symmetry of equals. Custom definitions of equality are usually  
044 \* based on an interface, such as Set or List, so that the implementation of equals  
045 \* can cast the object being tested for equality to the custom interface.  
046 \* AbstractCollectionDecorator does not implement such custom interfaces directly;  
047 \* they are implemented only in subclasses. Therefore, forwarding equals would break  
048 \* symmetry, as the forwarding object might consider itself equal to the object being  
049 \* tested, but the reverse could not be true. This behavior is consistent with the  
050 \* JDK's collection wrappers, such as {@link java.util.Collections#unmodifiableCollection(Collection)}.  
051 \* Use an interface-specific subclass of AbstractCollectionDecorator, such as  
052 \* AbstractListDecorator, to preserve equality behavior, or override equals directly.  
053 \* </p>  
054 \*  
055 \* @param <E> the type of the elements in the collection  
056 \* @since 3.0  
057 \*/  
058public abstract class AbstractCollectionDecorator<E>  
059 implements Collection<E>, Serializable {  
060  
061 /\*\* Serialization version \*/  
062 private static final long serialVersionUID = 6249888059822088500L;  
063  
064 /\*\* The collection being decorated \*/  
065 private Collection<E> collection;  
066  
067 /\*\*  
068 \* Constructor only used in deserialization, do not use otherwise.  
069 \* @since 3.1  
070 \*/  
071 protected AbstractCollectionDecorator() {  
072 super();  
073 }  
074  
075 /\*\*  
076 \* Constructor that wraps (not copies).  
077 \*  
078 \* @param coll the collection to decorate, must not be null  
079 \* @throws NullPointerException if the collection is null  
080 \*/  
081 protected AbstractCollectionDecorator(final Collection<E> coll) {  
082 if (coll == null) {  
083 throw new NullPointerException("Collection must not be null.");  
084 }  
085 this.collection = coll;  
086 }  
087  
088 /\*\*  
089 \* Gets the collection being decorated.  
090 \* All access to the decorated collection goes via this method.  
091 \*  
092 \* @return the decorated collection  
093 \*/  
094 protected Collection<E> decorated() {  
095 return collection;  
096 }  
097  
098 /\*\*  
099 \* Sets the collection being decorated.  
100 \* <p>  
101 \* <b>NOTE:</b> this method should only be used during deserialization  
102 \*  
103 \* @param coll the decorated collection  
104 \*/  
105 protected void setCollection(final Collection<E> coll) {  
106 this.collection = coll;  
107 }  
108  
109 //-----------------------------------------------------------------------  
110  
111 @Override  
112 public boolean add(final E object) {  
113 return decorated().add(object);  
114 }  
115  
116 @Override  
117 public boolean addAll(final Collection<? extends E> coll) {  
118 return decorated().addAll(coll);  
119 }  
120  
121 @Override  
122 public void clear() {  
123 decorated().clear();  
124 }  
125  
126 @Override  
127 public boolean contains(final Object object) {  
128 return decorated().contains(object);  
129 }  
130  
131 @Override  
132 public boolean isEmpty() {  
133 return decorated().isEmpty();  
134 }  
135  
136 @Override  
137 public Iterator<E> iterator() {  
138 return decorated().iterator();  
139 }  
140  
141 @Override  
142 public boolean remove(final Object object) {  
143 return decorated().remove(object);  
144 }  
145  
146 @Override  
147 public int size() {  
148 return decorated().size();  
149 }  
150  
151 @Override  
152 public Object[] toArray() {  
153 return decorated().toArray();  
154 }  
155  
156 @Override  
157 public <T> T[] toArray(final T[] object) {  
158 return decorated().toArray(object);  
159 }  
160  
161 @Override  
162 public boolean containsAll(final Collection<?> coll) {  
163 return decorated().containsAll(coll);  
164 }  
165  
166 /\*\*  
167 \* @since 4.4  
168 \*/  
169 @Override  
170 public boolean removeIf(final Predicate<? super E> filter) {  
171 return decorated().removeIf(filter);  
172 }  
173  
174 @Override  
175 public boolean removeAll(final Collection<?> coll) {  
176 return decorated().removeAll(coll);  
177 }  
178  
179 @Override  
180 public boolean retainAll(final Collection<?> coll) {  
181 return decorated().retainAll(coll);  
182 }  
183  
184 @Override  
185 public String toString() {  
186 return decorated().toString();  
187 }  
188  
189}