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.set;  
018  
019import java.util.Iterator;  
020import java.util.NavigableSet;  
021  
022import org.apache.commons.collections4.Predicate;  
023  
024/\*\*  
025 \* Decorates another <code>NavigableSet</code> to validate that all additions  
026 \* match a specified predicate.  
027 \* <p>  
028 \* This set exists to provide validation for the decorated set.  
029 \* It is normally created to decorate an empty set.  
030 \* If an object cannot be added to the set, an IllegalArgumentException is thrown.  
031 \* </p>  
032 \* <p>  
033 \* One usage would be to ensure that no null entries are added to the set.  
034 \* </p>  
035 \* <pre>  
036 \* NavigableSet set =  
037 \* PredicatedSortedSet.predicatedNavigableSet(new TreeSet(),  
038 \* NotNullPredicate.notNullPredicate());  
039 \* </pre>  
040 \*  
041 \* @param <E> the type of the elements in this set  
042 \* @since 4.1  
043 \*/  
044public class PredicatedNavigableSet<E> extends PredicatedSortedSet<E> implements NavigableSet<E> {  
045  
046 /\*\* Serialization version \*/  
047 private static final long serialVersionUID = 20150528L;  
048  
049 /\*\*  
050 \* Factory method to create a predicated (validating) navigable set.  
051 \* <p>  
052 \* If there are any elements already in the set being decorated, they  
053 \* are validated.  
054 \*  
055 \* @param <E> the element type  
056 \* @param set the set to decorate, must not be null  
057 \* @param predicate the predicate to use for validation, must not be null  
058 \* @return a new predicated navigable set.  
059 \* @throws NullPointerException if set or predicate is null  
060 \* @throws IllegalArgumentException if the set contains invalid elements  
061 \* @since 4.0  
062 \*/  
063 public static <E> PredicatedNavigableSet<E> predicatedNavigableSet(final NavigableSet<E> set,  
064 final Predicate<? super E> predicate) {  
065 return new PredicatedNavigableSet<>(set, predicate);  
066 }  
067  
068 //-----------------------------------------------------------------------  
069 /\*\*  
070 \* Constructor that wraps (not copies).  
071 \* <p>  
072 \* If there are any elements already in the set being decorated, they  
073 \* are validated.  
074 \*  
075 \* @param set the set to decorate, must not be null  
076 \* @param predicate the predicate to use for validation, must not be null  
077 \* @throws NullPointerException if set or predicate is null  
078 \* @throws IllegalArgumentException if the set contains invalid elements  
079 \*/  
080 protected PredicatedNavigableSet(final NavigableSet<E> set, final Predicate<? super E> predicate) {  
081 super(set, predicate);  
082 }  
083  
084 /\*\*  
085 \* Gets the navigable set being decorated.  
086 \*  
087 \* @return the decorated navigable set  
088 \*/  
089 @Override  
090 protected NavigableSet<E> decorated() {  
091 return (NavigableSet<E>) super.decorated();  
092 }  
093  
094 //-----------------------------------------------------------------------  
095  
096 @Override  
097 public E lower(final E e) {  
098 return decorated().lower(e);  
099 }  
100  
101 @Override  
102 public E floor(final E e) {  
103 return decorated().floor(e);  
104 }  
105  
106 @Override  
107 public E ceiling(final E e) {  
108 return decorated().ceiling(e);  
109 }  
110  
111 @Override  
112 public E higher(final E e) {  
113 return decorated().higher(e);  
114 }  
115  
116 @Override  
117 public E pollFirst() {  
118 return decorated().pollFirst();  
119 }  
120  
121 @Override  
122 public E pollLast() {  
123 return decorated().pollLast();  
124 }  
125  
126 @Override  
127 public NavigableSet<E> descendingSet() {  
128 return predicatedNavigableSet(decorated().descendingSet(), predicate);  
129 }  
130  
131 @Override  
132 public Iterator<E> descendingIterator() {  
133 return decorated().descendingIterator();  
134 }  
135  
136 @Override  
137 public NavigableSet<E> subSet(final E fromElement, final boolean fromInclusive, final E toElement,  
138 final boolean toInclusive) {  
139 final NavigableSet<E> sub = decorated().subSet(fromElement, fromInclusive, toElement, toInclusive);  
140 return predicatedNavigableSet(sub, predicate);  
141 }  
142  
143 @Override  
144 public NavigableSet<E> headSet(final E toElement, final boolean inclusive) {  
145 final NavigableSet<E> head = decorated().headSet(toElement, inclusive);  
146 return predicatedNavigableSet(head, predicate);  
147 }  
148  
149 @Override  
150 public NavigableSet<E> tailSet(final E fromElement, final boolean inclusive) {  
151 final NavigableSet<E> tail = decorated().tailSet(fromElement, inclusive);  
152 return predicatedNavigableSet(tail, predicate);  
153 }  
154  
155}