001/\*  
002 \* Licensed to the Apache Software Foundation (ASF) under one  
003 \* or more contributor license agreements. See the NOTICE file  
004 \* distributed with this work for additional information  
005 \* regarding copyright ownership. The ASF licenses this file  
006 \* to you under the Apache License, Version 2.0 (the  
007 \* "License"); you may not use this file except in compliance  
008 \* with the License. You may obtain a copy of the License at  
009 \*  
010 \* http://www.apache.org/licenses/LICENSE-2.0  
011 \*  
012 \* Unless required by applicable law or agreed to in writing, software  
013 \* distributed under the License is distributed on an "AS IS" BASIS,  
014 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
015 \* See the License for the specific language governing permissions and  
016 \* limitations under the License.  
017 \*/  
018package org.apache.commons.collections4.iterators;  
019  
020import java.util.Iterator;  
021import java.util.NoSuchElementException;  
022  
023/\*\*  
024 \* Decorates an iterator to support one-element lookahead while iterating.  
025 \* <p>  
026 \* The decorator supports the removal operation, but an {@link IllegalStateException}  
027 \* will be thrown if {@link #remove()} is called directly after a call to  
028 \* {@link #peek()} or {@link #element()}.  
029 \*  
030 \* @since 4.0  
031 \*/  
032public class PeekingIterator<E> implements Iterator<E> {  
033  
034 /\*\* The iterator being decorated. \*/  
035 private final Iterator<? extends E> iterator;  
036  
037 /\*\* Indicates that the decorated iterator is exhausted. \*/  
038 private boolean exhausted = false;  
039  
040 /\*\* Indicates if the lookahead slot is filled. \*/  
041 private boolean slotFilled = false;  
042  
043 /\*\* The current slot for lookahead. \*/  
044 private E slot;  
045  
046 //-----------------------------------------------------------------------  
047 /\*\*  
048 \* Decorates the specified iterator to support one-element lookahead.  
049 \* <p>  
050 \* If the iterator is already a {@link PeekingIterator} it is returned directly.  
051 \*  
052 \* @param <E> the element type  
053 \* @param iterator the iterator to decorate  
054 \* @return a new peeking iterator  
055 \* @throws NullPointerException if the iterator is null  
056 \*/  
057 public static <E> PeekingIterator<E> peekingIterator(final Iterator<? extends E> iterator) {  
058 if (iterator == null) {  
059 throw new NullPointerException("Iterator must not be null");  
060 }  
061 if (iterator instanceof PeekingIterator<?>) {  
062 @SuppressWarnings("unchecked") // safe cast  
063 final PeekingIterator<E> it = (PeekingIterator<E>) iterator;  
064 return it;  
065 }  
066 return new PeekingIterator<>(iterator);  
067 }  
068  
069 //-----------------------------------------------------------------------  
070  
071 /\*\*  
072 \* Constructor.  
073 \*  
074 \* @param iterator the iterator to decorate  
075 \*/  
076 public PeekingIterator(final Iterator<? extends E> iterator) {  
077 this.iterator = iterator;  
078 }  
079  
080 private void fill() {  
081 if (exhausted || slotFilled) {  
082 return;  
083 }  
084 if (iterator.hasNext()) {  
085 slot = iterator.next();  
086 slotFilled = true;  
087 } else {  
088 exhausted = true;  
089 slot = null;  
090 slotFilled = false;  
091 }  
092 }  
093  
094 //-----------------------------------------------------------------------  
095 @Override  
096 public boolean hasNext() {  
097 if (exhausted) {  
098 return false;  
099 }  
100 return slotFilled || iterator.hasNext();  
101 }  
102  
103 /\*\*  
104 \* Returns the next element in iteration without advancing the underlying iterator.  
105 \* If the iterator is already exhausted, null will be returned.  
106 \* <p>  
107 \* Note: this method does not throw a {@link NoSuchElementException} if the iterator  
108 \* is already exhausted. If you want such a behavior, use {@link #element()} instead.  
109 \* <p>  
110 \* The rationale behind this is to follow the {@link java.util.Queue} interface  
111 \* which uses the same terminology.  
112 \*  
113 \* @return the next element from the iterator  
114 \*/  
115 public E peek() {  
116 fill();  
117 return exhausted ? null : slot;  
118 }  
119  
120 /\*\*  
121 \* Returns the next element in iteration without advancing the underlying iterator.  
122 \* If the iterator is already exhausted, null will be returned.  
123 \*  
124 \* @return the next element from the iterator  
125 \* @throws NoSuchElementException if the iterator is already exhausted according to {@link #hasNext()}  
126 \*/  
127 public E element() {  
128 fill();  
129 if (exhausted) {  
130 throw new NoSuchElementException();  
131 }  
132 return slot;  
133 }  
134  
135 @Override  
136 public E next() {  
137 if (!hasNext()) {  
138 throw new NoSuchElementException();  
139 }  
140 final E x = slotFilled ? slot : iterator.next();  
141 // reset the lookahead slot  
142 slot = null;  
143 slotFilled = false;  
144 return x;  
145 }  
146  
147 /\*\*  
148 \* {@inheritDoc}  
149 \*  
150 \* @throws IllegalStateException if {@link #peek()} or {@link #element()} has been called  
151 \* prior to the call to {@link #remove()}  
152 \*/  
153 @Override  
154 public void remove() {  
155 if (slotFilled) {  
156 throw new IllegalStateException("peek() or element() called before remove()");  
157 }  
158 iterator.remove();  
159 }  
160  
161}