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.ArrayDeque;  
021import java.util.Deque;  
022import java.util.Iterator;  
023  
024/\*\*  
025 \* Decorates an iterator to support pushback of elements.  
026 \* <p>  
027 \* The decorator stores the pushed back elements in a LIFO manner: the last element  
028 \* that has been pushed back, will be returned as the next element in a call to {@link #next()}.  
029 \* <p>  
030 \* The decorator does not support the removal operation. Any call to {@link #remove()} will  
031 \* result in an {@link UnsupportedOperationException}.  
032 \*  
033 \* @since 4.0  
034 \*/  
035public class PushbackIterator<E> implements Iterator<E> {  
036  
037 /\*\* The iterator being decorated. \*/  
038 private final Iterator<? extends E> iterator;  
039  
040 /\*\* The LIFO queue containing the pushed back items. \*/  
041 private final Deque<E> items = new ArrayDeque<>();  
042  
043 //-----------------------------------------------------------------------  
044 /\*\*  
045 \* Decorates the specified iterator to support one-element lookahead.  
046 \* <p>  
047 \* If the iterator is already a {@link PushbackIterator} it is returned directly.  
048 \*  
049 \* @param <E> the element type  
050 \* @param iterator the iterator to decorate  
051 \* @return a new peeking iterator  
052 \* @throws NullPointerException if the iterator is null  
053 \*/  
054 public static <E> PushbackIterator<E> pushbackIterator(final Iterator<? extends E> iterator) {  
055 if (iterator == null) {  
056 throw new NullPointerException("Iterator must not be null");  
057 }  
058 if (iterator instanceof PushbackIterator<?>) {  
059 @SuppressWarnings("unchecked") // safe cast  
060 final PushbackIterator<E> it = (PushbackIterator<E>) iterator;  
061 return it;  
062 }  
063 return new PushbackIterator<>(iterator);  
064 }  
065  
066 //-----------------------------------------------------------------------  
067  
068 /\*\*  
069 \* Constructor.  
070 \*  
071 \* @param iterator the iterator to decorate  
072 \*/  
073 public PushbackIterator(final Iterator<? extends E> iterator) {  
074 super();  
075 this.iterator = iterator;  
076 }  
077  
078 /\*\*  
079 \* Push back the given element to the iterator.  
080 \* <p>  
081 \* Calling {@link #next()} immediately afterwards will return exactly this element.  
082 \*  
083 \* @param item the element to push back to the iterator  
084 \*/  
085 public void pushback(final E item) {  
086 items.push(item);  
087 }  
088  
089 @Override  
090 public boolean hasNext() {  
091 return !items.isEmpty() || iterator.hasNext();  
092 }  
093  
094 @Override  
095 public E next() {  
096 return !items.isEmpty() ? items.pop() : iterator.next();  
097 }  
098  
099 /\*\*  
100 \* This iterator will always throw an {@link UnsupportedOperationException}.  
101 \*  
102 \* @throws UnsupportedOperationException always  
103 \*/  
104 @Override  
105 public void remove() {  
106 throw new UnsupportedOperationException();  
107 }  
108  
109}