001/\*  
002 \* Licensed to the Apache Software Foundation (ASF) under one or more  
003 \* contributor license agreements. See the NOTICE file distributed with this  
004 \* work for additional information regarding copyright ownership. The ASF  
005 \* licenses this file to You under the Apache License, Version 2.0 (the  
006 \* "License"); you may not use this file except in compliance with the License.  
007 \* You may obtain a copy of the License at  
008 \* http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law  
009 \* or agreed to in writing, software distributed under the License is  
010 \* distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY  
011 \* KIND, either express or implied. See the License for the specific language  
012 \* governing permissions and limitations under the License.  
013 \*/  
014package org.apache.commons.collections4.iterators;  
015  
016import java.util.Iterator;  
017import java.util.NoSuchElementException;  
018  
019/\*\*  
020 \* Decorates another iterator to return elements in a specific range.  
021 \* <p>  
022 \* The decorated iterator is bounded in the range [offset, offset+max).  
023 \* The {@code offset} corresponds to the position of the first element to  
024 \* be returned from the decorated iterator, and {@code max} is the maximum  
025 \* number of elements to be returned at most.  
026 \* <p>  
027 \* In case an offset parameter other than 0 is provided, the decorated  
028 \* iterator is immediately advanced to this position, skipping all elements  
029 \* before that position.  
030 \*  
031 \* @since 4.1  
032 \*/  
033public class BoundedIterator<E> implements Iterator<E> {  
034  
035 /\*\* The iterator being decorated. \*/  
036 private final Iterator<? extends E> iterator;  
037  
038 /\*\* The offset to bound the first element return \*/  
039 private final long offset;  
040  
041 /\*\* The max number of elements to return \*/  
042 private final long max;  
043  
044 /\*\* The position of the current element \*/  
045 private long pos;  
046  
047 //-----------------------------------------------------------------------  
048  
049 /\*\*  
050 \* Decorates the specified iterator to return at most the given number of elements,  
051 \* skipping all elements until the iterator reaches the position at {@code offset}.  
052 \* <p>  
053 \* The iterator is immediately advanced until it reaches the position at {@code offset},  
054 \* incurring O(n) time.  
055 \*  
056 \* @param iterator the iterator to be decorated  
057 \* @param offset the index of the first element of the decorated iterator to return  
058 \* @param max the maximum number of elements of the decorated iterator to return  
059 \* @throws NullPointerException if iterator is null  
060 \* @throws IllegalArgumentException if either offset or max is negative  
061 \*/  
062 public BoundedIterator(final Iterator<? extends E> iterator, final long offset, final long max) {  
063 if (iterator == null) {  
064 throw new NullPointerException("Iterator must not be null");  
065 }  
066 if (offset < 0) {  
067 throw new IllegalArgumentException("Offset parameter must not be negative.");  
068 }  
069 if (max < 0) {  
070 throw new IllegalArgumentException("Max parameter must not be negative.");  
071 }  
072  
073 this.iterator = iterator;  
074 this.offset = offset;  
075 this.max = max;  
076 pos = 0;  
077 init();  
078 }  
079  
080 /\*\*  
081 \* Advances the underlying iterator to the beginning of the bounded range.  
082 \*/  
083 private void init() {  
084 while (pos < offset && iterator.hasNext()) {  
085 iterator.next();  
086 pos++;  
087 }  
088 }  
089  
090 //-----------------------------------------------------------------------  
091  
092 @Override  
093 public boolean hasNext() {  
094 if (!checkBounds()) {  
095 return false;  
096 }  
097 return iterator.hasNext();  
098 }  
099  
100 /\*\*  
101 \* Checks whether the iterator is still within its bounded range.  
102 \* @return {@code true} if the iterator is within its bounds, {@code false} otherwise  
103 \*/  
104 private boolean checkBounds() {  
105 if (pos - offset + 1 > max) {  
106 return false;  
107 }  
108 return true;  
109 }  
110  
111 @Override  
112 public E next() {  
113 if (!checkBounds()) {  
114 throw new NoSuchElementException();  
115 }  
116 final E next = iterator.next();  
117 pos++;  
118 return next;  
119 }  
120  
121 /\*\*  
122 \* {@inheritDoc}  
123 \* <p>  
124 \* In case an offset other than 0 was specified, the underlying iterator will be advanced  
125 \* to this position upon creation. A call to {@link #remove()} will still result in an  
126 \* {@link IllegalStateException} if no explicit call to {@link #next()} has been made prior  
127 \* to calling {@link #remove()}.  
128 \*/  
129 @Override  
130 public void remove() {  
131 if (pos <= offset) {  
132 throw new IllegalStateException("remove() can not be called before calling next()");  
133 }  
134 iterator.remove();  
135 }  
136}