/\*

\* Copyright (c) 1997, 2013, Oracle and/or its affiliates. All rights reserved.

\* ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.

\*/

package java.util;

import java.util.function.Consumer;

import java.util.function.Predicate;

import java.util.function.UnaryOperator;

/\*\*

\* Resizable-array implementation of the <tt>List</tt> interface. Implements

\* all optional list operations, and permits all elements, including

\* <tt>null</tt>. In addition to implementing the <tt>List</tt> interface,

\* this class provides methods to manipulate the size of the array that is

\* used internally to store the list. (This class is roughly equivalent to

\* <tt>Vector</tt>, except that it is unsynchronized.)

\*

\* <p>The <tt>size</tt>, <tt>isEmpty</tt>, <tt>get</tt>, <tt>set</tt>,

\* <tt>iterator</tt>, and <tt>listIterator</tt> operations run in constant

\* time. The <tt>add</tt> operation runs in <i>amortized constant time</i>,

\* that is, adding n elements requires O(n) time. All of the other operations

\* run in linear time (roughly speaking). The constant factor is low compared

\* to that for the <tt>LinkedList</tt> implementation.

\*

\* <p>Each <tt>ArrayList</tt> instance has a <i>capacity</i>. The capacity is

\* the size of the array used to store the elements in the list. It is always

\* at least as large as the list size. As elements are added to an ArrayList,

\* its capacity grows automatically. The details of the growth policy are not

\* specified beyond the fact that adding an element has constant amortized

\* time cost.

\*

\* <p>An application can increase the capacity of an <tt>ArrayList</tt> instance

\* before adding a large number of elements using the <tt>ensureCapacity</tt>

\* operation. This may reduce the amount of incremental reallocation.

\*

\* <p><strong>Note that this implementation is not synchronized.</strong>

\* If multiple threads access an <tt>ArrayList</tt> instance concurrently,

\* and at least one of the threads modifies the list structurally, it

\* <i>must</i> be synchronized externally. (A structural modification is

\* any operation that adds or deletes one or more elements, or explicitly

\* resizes the backing array; merely setting the value of an element is not

\* a structural modification.) This is typically accomplished by

\* synchronizing on some object that naturally encapsulates the list.

\*

\* If no such object exists, the list should be "wrapped" using the

\* {@link Collections#synchronizedList Collections.synchronizedList}

\* method. This is best done at creation time, to prevent accidental

\* unsynchronized access to the list:<pre>

\* List list = Collections.synchronizedList(new ArrayList(...));</pre>

\*

\* <p><a name="fail-fast">

\* The iterators returned by this class's {@link #iterator() iterator} and

\* {@link #listIterator(int) listIterator} methods are <em>fail-fast</em>:</a>

\* if the list is structurally modified at any time after the iterator is

\* created, in any way except through the iterator's own

\* {@link ListIterator#remove() remove} or

\* {@link ListIterator#add(Object) add} methods, the iterator will throw a

\* {@link ConcurrentModificationException}. Thus, in the face of

\* concurrent modification, the iterator fails quickly and cleanly, rather

\* than risking arbitrary, non-deterministic behavior at an undetermined

\* time in the future.

\*

\* <p>Note that the fail-fast behavior of an iterator cannot be guaranteed

\* as it is, generally speaking, impossible to make any hard guarantees in the

\* presence of unsynchronized concurrent modification. Fail-fast iterators

\* throw {@code ConcurrentModificationException} on a best-effort basis.

\* Therefore, it would be wrong to write a program that depended on this

\* exception for its correctness: <i>the fail-fast behavior of iterators

\* should be used only to detect bugs.</i>

\*

\* <p>This class is a member of the

\* <a href="{@docRoot}/../technotes/guides/collections/index.html">

\* Java Collections Framework</a>.

\*

\* @author Josh Bloch

\* @author Neal Gafter

\* @see Collection

\* @see List

\* @see LinkedList

\* @see Vector

\* @since 1.2

\*/

public class ArrayList<E> extends AbstractList<E>

implements List<E>, RandomAccess, Cloneable, java.io.Serializable

{

private static final long serialVersionUID = 8683452581122892189L;

/\*\*

\* Default initial capacity.

\*/

private static final int DEFAULT\_CAPACITY = 10;

/\*\*

\* Shared empty array instance used for empty instances.

\*/

private static final Object[] EMPTY\_ELEMENTDATA = {};

/\*\*

\* Shared empty array instance used for default sized empty instances. We

\* distinguish this from EMPTY\_ELEMENTDATA to know how much to inflate when

\* first element is added.

\*/

private static final Object[] DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA = {};

/\*\*

\* The array buffer into which the elements of the ArrayList are stored.

\* The capacity of the ArrayList is the length of this array buffer. Any

\* empty ArrayList with elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA

\* will be expanded to DEFAULT\_CAPACITY when the first element is added.

\*/

transient Object[] elementData; // non-private to simplify nested class access

/\*\*

\* The size of the ArrayList (the number of elements it contains).

\*

\* @serial

\*/

private int size;

/\*\*

\* Constructs an empty list with the specified initial capacity.

\*

\* @param initialCapacity the initial capacity of the list

\* @throws IllegalArgumentException if the specified initial capacity

\* is negative

\*/

public ArrayList(int initialCapacity) {

if (initialCapacity > 0) {

this.elementData = new Object[initialCapacity];

} else if (initialCapacity == 0) {

this.elementData = EMPTY\_ELEMENTDATA;

} else {

throw new IllegalArgumentException("Illegal Capacity: "+

initialCapacity);

}

}

/\*\*

\* Constructs an empty list with an initial capacity of ten.

\*/

public ArrayList() {

this.elementData = DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA;

}

/\*\*

\* Constructs a list containing the elements of the specified

\* collection, in the order they are returned by the collection's

\* iterator.

\*

\* @param c the collection whose elements are to be placed into this list

\* @throws NullPointerException if the specified collection is null

\*/

public ArrayList(Collection<? extends E> c) {

elementData = c.toArray();

if ((size = elementData.length) != 0) {

// c.toArray might (incorrectly) not return Object[] (see 6260652)

if (elementData.getClass() != Object[].class)

elementData = Arrays.copyOf(elementData, size, Object[].class);

} else {

// replace with empty array.

this.elementData = EMPTY\_ELEMENTDATA;

}

}

/\*\*

\* Trims the capacity of this <tt>ArrayList</tt> instance to be the

\* list's current size. An application can use this operation to minimize

\* the storage of an <tt>ArrayList</tt> instance.

\*/

public void trimToSize() {

modCount++;

if (size < elementData.length) {

elementData = (size == 0)

? EMPTY\_ELEMENTDATA

: Arrays.copyOf(elementData, size);

}

}

/\*\*

\* Increases the capacity of this <tt>ArrayList</tt> instance, if

\* necessary, to ensure that it can hold at least the number of elements

\* specified by the minimum capacity argument.

\*

\* @param minCapacity the desired minimum capacity

\*/

public void ensureCapacity(int minCapacity) {

int minExpand = (elementData != DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA)

// any size if not default element table

? 0

// larger than default for default empty table. It's already

// supposed to be at default size.

: DEFAULT\_CAPACITY;

if (minCapacity > minExpand) {

ensureExplicitCapacity(minCapacity);

}

}

private void ensureCapacityInternal(int minCapacity) {

if (elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA) {

minCapacity = Math.max(DEFAULT\_CAPACITY, minCapacity);

}

ensureExplicitCapacity(minCapacity);

}

private void ensureExplicitCapacity(int minCapacity) {

modCount++;

// overflow-conscious code

if (minCapacity - elementData.length > 0)

grow(minCapacity);

}

/\*\*

\* The maximum size of array to allocate.

\* Some VMs reserve some header words in an array.

\* Attempts to allocate larger arrays may result in

\* OutOfMemoryError: Requested array size exceeds VM limit

\*/

private static final int MAX\_ARRAY\_SIZE = Integer.MAX\_VALUE - 8;

/\*\*

\* Increases the capacity to ensure that it can hold at least the

\* number of elements specified by the minimum capacity argument.

\*

\* @param minCapacity the desired minimum capacity

\*/

private void grow(int minCapacity) {

// overflow-conscious code

int oldCapacity = elementData.length;

int newCapacity = oldCapacity + (oldCapacity >> 1);

if (newCapacity - minCapacity < 0)

newCapacity = minCapacity;

if (newCapacity - MAX\_ARRAY\_SIZE > 0)

newCapacity = hugeCapacity(minCapacity);

// minCapacity is usually close to size, so this is a win:

elementData = Arrays.copyOf(elementData, newCapacity);

}

private static int hugeCapacity(int minCapacity) {

if (minCapacity < 0) // overflow

throw new OutOfMemoryError();

return (minCapacity > MAX\_ARRAY\_SIZE) ?

Integer.MAX\_VALUE :

MAX\_ARRAY\_SIZE;

}

/\*\*

\* Returns the number of elements in this list.

\*

\* @return the number of elements in this list

\*/

public int size() {

return size;

}

/\*\*

\* Returns <tt>true</tt> if this list contains no elements.

\*

\* @return <tt>true</tt> if this list contains no elements

\*/

public boolean isEmpty() {

return size == 0;

}

/\*\*

\* Returns <tt>true</tt> if this list contains the specified element.

\* More formally, returns <tt>true</tt> if and only if this list contains

\* at least one element <tt>e</tt> such that

\* <tt>(o==null&nbsp;?&nbsp;e==null&nbsp;:&nbsp;o.equals(e))</tt>.

\*

\* @param o element whose presence in this list is to be tested

\* @return <tt>true</tt> if this list contains the specified element

\*/

public boolean contains(Object o) {

return indexOf(o) >= 0;

}

/\*\*

\* Returns the index of the first occurrence of the specified element

\* in this list, or -1 if this list does not contain the element.

\* More formally, returns the lowest index <tt>i</tt> such that

\* <tt>(o==null&nbsp;?&nbsp;get(i)==null&nbsp;:&nbsp;o.equals(get(i)))</tt>,

\* or -1 if there is no such index.

\*/

public int indexOf(Object o) {

if (o == null) {

for (int i = 0; i < size; i++)

if (elementData[i]==null)

return i;

} else {

for (int i = 0; i < size; i++)

if (o.equals(elementData[i]))

return i;

}

return -1;

}

/\*\*

\* Returns the index of the last occurrence of the specified element

\* in this list, or -1 if this list does not contain the element.

\* More formally, returns the highest index <tt>i</tt> such that

\* <tt>(o==null&nbsp;?&nbsp;get(i)==null&nbsp;:&nbsp;o.equals(get(i)))</tt>,

\* or -1 if there is no such index.

\*/

public int lastIndexOf(Object o) {

if (o == null) {

for (int i = size-1; i >= 0; i--)

if (elementData[i]==null)

return i;

} else {

for (int i = size-1; i >= 0; i--)

if (o.equals(elementData[i]))

return i;

}

return -1;

}

/\*\*

\* Returns a shallow copy of this <tt>ArrayList</tt> instance. (The

\* elements themselves are not copied.)

\*

\* @return a clone of this <tt>ArrayList</tt> instance

\*/

public Object clone() {

try {

ArrayList<?> v = (ArrayList<?>) super.clone();

v.elementData = Arrays.copyOf(elementData, size);

v.modCount = 0;

return v;

} catch (CloneNotSupportedException e) {

// this shouldn't happen, since we are Cloneable

throw new InternalError(e);

}

}

/\*\*

\* Returns an array containing all of the elements in this list

\* in proper sequence (from first to last element).

\*

\* <p>The returned array will be "safe" in that no references to it are

\* maintained by this list. (In other words, this method must allocate

\* a new array). The caller is thus free to modify the returned array.

\*

\* <p>This method acts as bridge between array-based and collection-based

\* APIs.

\*

\* @return an array containing all of the elements in this list in

\* proper sequence

\*/

public Object[] toArray() {

return Arrays.copyOf(elementData, size);

}

/\*\*

\* Returns an array containing all of the elements in this list in proper

\* sequence (from first to last element); the runtime type of the returned

\* array is that of the specified array. If the list fits in the

\* specified array, it is returned therein. Otherwise, a new array is

\* allocated with the runtime type of the specified array and the size of

\* this list.

\*

\* <p>If the list fits in the specified array with room to spare

\* (i.e., the array has more elements than the list), the element in

\* the array immediately following the end of the collection is set to

\* <tt>null</tt>. (This is useful in determining the length of the

\* list <i>only</i> if the caller knows that the list does not contain

\* any null elements.)

\*

\* @param a the array into which the elements of the list are to

\* be stored, if it is big enough; otherwise, a new array of the

\* same runtime type is allocated for this purpose.

\* @return an array containing the elements of the list

\* @throws ArrayStoreException if the runtime type of the specified array

\* is not a supertype of the runtime type of every element in

\* this list

\* @throws NullPointerException if the specified array is null

\*/

@SuppressWarnings("unchecked")

public <T> T[] toArray(T[] a) {

if (a.length < size)

// Make a new array of a's runtime type, but my contents:

return (T[]) Arrays.copyOf(elementData, size, a.getClass());

System.arraycopy(elementData, 0, a, 0, size);

if (a.length > size)

a[size] = null;

return a;

}

// Positional Access Operations

@SuppressWarnings("unchecked")

E elementData(int index) {

return (E) elementData[index];

}

/\*\*

\* Returns the element at the specified position in this list.

\*

\* @param index index of the element to return

\* @return the element at the specified position in this list

\* @throws IndexOutOfBoundsException {@inheritDoc}

\*/

public E get(int index) {

rangeCheck(index);

return elementData(index);

}

/\*\*

\* Replaces the element at the specified position in this list with

\* the specified element.

\*

\* @param index index of the element to replace

\* @param element element to be stored at the specified position

\* @return the element previously at the specified position

\* @throws IndexOutOfBoundsException {@inheritDoc}

\*/

public E set(int index, E element) {

rangeCheck(index);

E oldValue = elementData(index);

elementData[index] = element;

return oldValue;

}

/\*\*

\* Appends the specified element to the end of this list.

\*

\* @param e element to be appended to this list

\* @return <tt>true</tt> (as specified by {@link Collection#add})

\*/

public boolean add(E e) {

ensureCapacityInternal(size + 1); // Increments modCount!!

elementData[size++] = e;

return true;

}

/\*\*

\* Inserts the specified element at the specified position in this

\* list. Shifts the element currently at that position (if any) and

\* any subsequent elements to the right (adds one to their indices).

\*

\* @param index index at which the specified element is to be inserted

\* @param element element to be inserted

\* @throws IndexOutOfBoundsException {@inheritDoc}

\*/

public void add(int index, E element) {

rangeCheckForAdd(index);

ensureCapacityInternal(size + 1); // Increments modCount!!

System.arraycopy(elementData, index, elementData, index + 1,

size - index);

elementData[index] = element;

size++;

}

/\*\*

\* Removes the element at the specified position in this list.

\* Shifts any subsequent elements to the left (subtracts one from their

\* indices).

\*

\* @param index the index of the element to be removed

\* @return the element that was removed from the list

\* @throws IndexOutOfBoundsException {@inheritDoc}

\*/

public E remove(int index) {

rangeCheck(index);

modCount++;

E oldValue = elementData(index);

int numMoved = size - index - 1;

if (numMoved > 0)

System.arraycopy(elementData, index+1, elementData, index,

numMoved);

elementData[--size] = null; // clear to let GC do its work

return oldValue;

}

/\*\*

\* Removes the first occurrence of the specified element from this list,

\* if it is present. If the list does not contain the element, it is

\* unchanged. More formally, removes the element with the lowest index

\* <tt>i</tt> such that

\* <tt>(o==null&nbsp;?&nbsp;get(i)==null&nbsp;:&nbsp;o.equals(get(i)))</tt>

\* (if such an element exists). Returns <tt>true</tt> if this list

\* contained the specified element (or equivalently, if this list

\* changed as a result of the call).

\*

\* @param o element to be removed from this list, if present

\* @return <tt>true</tt> if this list contained the specified element

\*/

public boolean remove(Object o) {

if (o == null) {

for (int index = 0; index < size; index++)

if (elementData[index] == null) {

fastRemove(index);

return true;

}

} else {

for (int index = 0; index < size; index++)

if (o.equals(elementData[index])) {

fastRemove(index);

return true;

}

}

return false;

}

/\*

\* Private remove method that skips bounds checking and does not

\* return the value removed.

\*/

private void fastRemove(int index) {

modCount++;

int numMoved = size - index - 1;

if (numMoved > 0)

System.arraycopy(elementData, index+1, elementData, index,

numMoved);

elementData[--size] = null; // clear to let GC do its work

}

/\*\*

\* Removes all of the elements from this list. The list will

\* be empty after this call returns.

\*/

public void clear() {

modCount++;

// clear to let GC do its work

for (int i = 0; i < size; i++)

elementData[i] = null;

size = 0;

}

/\*\*

\* Appends all of the elements in the specified collection to the end of

\* this list, in the order that they are returned by the

\* specified collection's Iterator. The behavior of this operation is

\* undefined if the specified collection is modified while the operation

\* is in progress. (This implies that the behavior of this call is

\* undefined if the specified collection is this list, and this

\* list is nonempty.)

\*

\* @param c collection containing elements to be added to this list

\* @return <tt>true</tt> if this list changed as a result of the call

\* @throws NullPointerException if the specified collection is null

\*/

public boolean addAll(Collection<? extends E> c) {

Object[] a = c.toArray();

int numNew = a.length;

ensureCapacityInternal(size + numNew); // Increments modCount

System.arraycopy(a, 0, elementData, size, numNew);

size += numNew;

return numNew != 0;

}

/\*\*

\* Inserts all of the elements in the specified collection into this

\* list, starting at the specified position. Shifts the element

\* currently at that position (if any) and any subsequent elements to

\* the right (increases their indices). The new elements will appear

\* in the list in the order that they are returned by the

\* specified collection's iterator.

\*

\* @param index index at which to insert the first element from the

\* specified collection

\* @param c collection containing elements to be added to this list

\* @return <tt>true</tt> if this list changed as a result of the call

\* @throws IndexOutOfBoundsException {@inheritDoc}

\* @throws NullPointerException if the specified collection is null

\*/

public boolean addAll(int index, Collection<? extends E> c) {

rangeCheckForAdd(index);

Object[] a = c.toArray();

int numNew = a.length;

ensureCapacityInternal(size + numNew); // Increments modCount

int numMoved = size - index;

if (numMoved > 0)

System.arraycopy(elementData, index, elementData, index + numNew,

numMoved);

System.arraycopy(a, 0, elementData, index, numNew);

size += numNew;

return numNew != 0;

}

/\*\*

\* Removes from this list all of the elements whose index is between

\* {@code fromIndex}, inclusive, and {@code toIndex}, exclusive.

\* Shifts any succeeding elements to the left (reduces their index).

\* This call shortens the list by {@code (toIndex - fromIndex)} elements.

\* (If {@code toIndex==fromIndex}, this operation has no effect.)

\*

\* @throws IndexOutOfBoundsException if {@code fromIndex} or

\* {@code toIndex} is out of range

\* ({@code fromIndex < 0 ||

\* fromIndex >= size() ||

\* toIndex > size() ||

\* toIndex < fromIndex})

\*/

protected void removeRange(int fromIndex, int toIndex) {

modCount++;

int numMoved = size - toIndex;

System.arraycopy(elementData, toIndex, elementData, fromIndex,

numMoved);

// clear to let GC do its work

int newSize = size - (toIndex-fromIndex);

for (int i = newSize; i < size; i++) {

elementData[i] = null;

}

size = newSize;

}

/\*\*

\* Checks if the given index is in range. If not, throws an appropriate

\* runtime exception. This method does \*not\* check if the index is

\* negative: It is always used immediately prior to an array access,

\* which throws an ArrayIndexOutOfBoundsException if index is negative.

\*/

private void rangeCheck(int index) {

if (index >= size)

throw new IndexOutOfBoundsException(outOfBoundsMsg(index));

}

/\*\*

\* A version of rangeCheck used by add and addAll.

\*/

private void rangeCheckForAdd(int index) {

if (index > size || index < 0)

throw new IndexOutOfBoundsException(outOfBoundsMsg(index));

}

/\*\*

\* Constructs an IndexOutOfBoundsException detail message.

\* Of the many possible refactorings of the error handling code,

\* this "outlining" performs best with both server and client VMs.

\*/

private String outOfBoundsMsg(int index) {

return "Index: "+index+", Size: "+size;

}

/\*\*

\* Removes from this list all of its elements that are contained in the

\* specified collection.

\*

\* @param c collection containing elements to be removed from this list

\* @return {@code true} if this list changed as a result of the call

\* @throws ClassCastException if the class of an element of this list

\* is incompatible with the specified collection

\* (<a href="Collection.html#optional-restrictions">optional</a>)

\* @throws NullPointerException if this list contains a null element and the

\* specified collection does not permit null elements

\* (<a href="Collection.html#optional-restrictions">optional</a>),

\* or if the specified collection is null

\* @see Collection#contains(Object)

\*/

public boolean removeAll(Collection<?> c) {

Objects.requireNonNull(c);

return batchRemove(c, false);

}

/\*\*

\* Retains only the elements in this list that are contained in the

\* specified collection. In other words, removes from this list all

\* of its elements that are not contained in the specified collection.

\*

\* @param c collection containing elements to be retained in this list

\* @return {@code true} if this list changed as a result of the call

\* @throws ClassCastException if the class of an element of this list

\* is incompatible with the specified collection

\* (<a href="Collection.html#optional-restrictions">optional</a>)

\* @throws NullPointerException if this list contains a null element and the

\* specified collection does not permit null elements

\* (<a href="Collection.html#optional-restrictions">optional</a>),

\* or if the specified collection is null

\* @see Collection#contains(Object)

\*/

public boolean retainAll(Collection<?> c) {

Objects.requireNonNull(c);

return batchRemove(c, true);

}

private boolean batchRemove(Collection<?> c, boolean complement) {

final Object[] elementData = this.elementData;

int r = 0, w = 0;

boolean modified = false;

try {

for (; r < size; r++)

if (c.contains(elementData[r]) == complement)

elementData[w++] = elementData[r];

} finally {

// Preserve behavioral compatibility with AbstractCollection,

// even if c.contains() throws.

if (r != size) {

System.arraycopy(elementData, r,

elementData, w,

size - r);

w += size - r;

}

if (w != size) {

// clear to let GC do its work

for (int i = w; i < size; i++)

elementData[i] = null;

modCount += size - w;

size = w;

modified = true;

}

}

return modified;

}

/\*\*

\* Save the state of the <tt>ArrayList</tt> instance to a stream (that

\* is, serialize it).

\*

\* @serialData The length of the array backing the <tt>ArrayList</tt>

\* instance is emitted (int), followed by all of its elements

\* (each an <tt>Object</tt>) in the proper order.

\*/

private void writeObject(java.io.ObjectOutputStream s)

throws java.io.IOException{

// Write out element count, and any hidden stuff

int expectedModCount = modCount;

s.defaultWriteObject();

// Write out size as capacity for behavioural compatibility with clone()

s.writeInt(size);

// Write out all elements in the proper order.

for (int i=0; i<size; i++) {

s.writeObject(elementData[i]);

}

if (modCount != expectedModCount) {

throw new ConcurrentModificationException();

}

}

/\*\*

\* Reconstitute the <tt>ArrayList</tt> instance from a stream (that is,

\* deserialize it).

\*/

private void readObject(java.io.ObjectInputStream s)

throws java.io.IOException, ClassNotFoundException {

elementData = EMPTY\_ELEMENTDATA;

// Read in size, and any hidden stuff

s.defaultReadObject();

// Read in capacity

s.readInt(); // ignored

if (size > 0) {

// be like clone(), allocate array based upon size not capacity

ensureCapacityInternal(size);

Object[] a = elementData;

// Read in all elements in the proper order.

for (int i=0; i<size; i++) {

a[i] = s.readObject();

}

}

}

/\*\*

\* Returns a list iterator over the elements in this list (in proper

\* sequence), starting at the specified position in the list.

\* The specified index indicates the first element that would be

\* returned by an initial call to {@link ListIterator#next next}.

\* An initial call to {@link ListIterator#previous previous} would

\* return the element with the specified index minus one.

\*

\* <p>The returned list iterator is <a href="#fail-fast"><i>fail-fast</i></a>.

\*

\* @throws IndexOutOfBoundsException {@inheritDoc}

\*/

public ListIterator<E> listIterator(int index) {

if (index < 0 || index > size)

throw new IndexOutOfBoundsException("Index: "+index);

return new ListItr(index);

}

/\*\*

\* Returns a list iterator over the elements in this list (in proper

\* sequence).

\*

\* <p>The returned list iterator is <a href="#fail-fast"><i>fail-fast</i></a>.

\*

\* @see #listIterator(int)

\*/

public ListIterator<E> listIterator() {

return new ListItr(0);

}

/\*\*

\* Returns an iterator over the elements in this list in proper sequence.

\*

\* <p>The returned iterator is <a href="#fail-fast"><i>fail-fast</i></a>.

\*

\* @return an iterator over the elements in this list in proper sequence

\*/

public Iterator<E> iterator() {

return new Itr();

}

/\*\*

\* An optimized version of AbstractList.Itr

\*/

private class Itr implements Iterator<E> {

int cursor; // index of next element to return

int lastRet = -1; // index of last element returned; -1 if no such

int expectedModCount = modCount;

public boolean hasNext() {

return cursor != size;

}

@SuppressWarnings("unchecked")

public E next() {

checkForComodification();

int i = cursor;

if (i >= size)

throw new NoSuchElementException();

Object[] elementData = ArrayList.this.elementData;

if (i >= elementData.length)

throw new ConcurrentModificationException();

cursor = i + 1;

return (E) elementData[lastRet = i];

}

public void remove() {

if (lastRet < 0)

throw new IllegalStateException();

checkForComodification();

try {

ArrayList.this.remove(lastRet);

cursor = lastRet;

lastRet = -1;

expectedModCount = modCount;

} catch (IndexOutOfBoundsException ex) {

throw new ConcurrentModificationException();

}

}

@Override

@SuppressWarnings("unchecked")

public void forEachRemaining(Consumer<? super E> consumer) {

Objects.requireNonNull(consumer);

final int size = ArrayList.this.size;

int i = cursor;

if (i >= size) {

return;

}

final Object[] elementData = ArrayList.this.elementData;

if (i >= elementData.length) {

throw new ConcurrentModificationException();

}

while (i != size && modCount == expectedModCount) {

consumer.accept((E) elementData[i++]);

}

// update once at end of iteration to reduce heap write traffic

cursor = i;

lastRet = i - 1;

checkForComodification();

}

final void checkForComodification() {

if (modCount != expectedModCount)

throw new ConcurrentModificationException();

}

}

/\*\*

\* An optimized version of AbstractList.ListItr

\*/

private class ListItr extends Itr implements ListIterator<E> {

ListItr(int index) {

super();

cursor = index;

}

public boolean hasPrevious() {

return cursor != 0;

}

public int nextIndex() {

return cursor;

}

public int previousIndex() {

return cursor - 1;

}

@SuppressWarnings("unchecked")

public E previous() {

checkForComodification();

int i = cursor - 1;

if (i < 0)

throw new NoSuchElementException();

Object[] elementData = ArrayList.this.elementData;

if (i >= elementData.length)

throw new ConcurrentModificationException();

cursor = i;

return (E) elementData[lastRet = i];

}

public void set(E e) {

if (lastRet < 0)

throw new IllegalStateException();

checkForComodification();

try {

ArrayList.this.set(lastRet, e);

} catch (IndexOutOfBoundsException ex) {

throw new ConcurrentModificationException();

}

}

public void add(E e) {

checkForComodification();

try {

int i = cursor;

ArrayList.this.add(i, e);

cursor = i + 1;

lastRet = -1;

expectedModCount = modCount;

} catch (IndexOutOfBoundsException ex) {

throw new ConcurrentModificationException();

}

}

}

/\*\*

\* Returns a view of the portion of this list between the specified

\* {@code fromIndex}, inclusive, and {@code toIndex}, exclusive. (If

\* {@code fromIndex} and {@code toIndex} are equal, the returned list is

\* empty.) The returned list is backed by this list, so non-structural

\* changes in the returned list are reflected in this list, and vice-versa.

\* The returned list supports all of the optional list operations.

\*

\* <p>This method eliminates the need for explicit range operations (of

\* the sort that commonly exist for arrays). Any operation that expects

\* a list can be used as a range operation by passing a subList view

\* instead of a whole list. For example, the following idiom

\* removes a range of elements from a list:

\* <pre>

\* list.subList(from, to).clear();

\* </pre>

\* Similar idioms may be constructed for {@link #indexOf(Object)} and

\* {@link #lastIndexOf(Object)}, and all of the algorithms in the

\* {@link Collections} class can be applied to a subList.

\*

\* <p>The semantics of the list returned by this method become undefined if

\* the backing list (i.e., this list) is <i>structurally modified</i> in

\* any way other than via the returned list. (Structural modifications are

\* those that change the size of this list, or otherwise perturb it in such

\* a fashion that iterations in progress may yield incorrect results.)

\*

\* @throws IndexOutOfBoundsException {@inheritDoc}

\* @throws IllegalArgumentException {@inheritDoc}

\*/

public List<E> subList(int fromIndex, int toIndex) {

subListRangeCheck(fromIndex, toIndex, size);

return new SubList(this, 0, fromIndex, toIndex);

}

static void subListRangeCheck(int fromIndex, int toIndex, int size) {

if (fromIndex < 0)

throw new IndexOutOfBoundsException("fromIndex = " + fromIndex);

if (toIndex > size)

throw new IndexOutOfBoundsException("toIndex = " + toIndex);

if (fromIndex > toIndex)

throw new IllegalArgumentException("fromIndex(" + fromIndex +

") > toIndex(" + toIndex + ")");

}

private class SubList extends AbstractList<E> implements RandomAccess {

private final AbstractList<E> parent;

private final int parentOffset;

private final int offset;

int size;

SubList(AbstractList<E> parent,

int offset, int fromIndex, int toIndex) {

this.parent = parent;

this.parentOffset = fromIndex;

this.offset = offset + fromIndex;

this.size = toIndex - fromIndex;

this.modCount = ArrayList.this.modCount;

}

public E set(int index, E e) {

rangeCheck(index);

checkForComodification();

E oldValue = ArrayList.this.elementData(offset + index);

ArrayList.this.elementData[offset + index] = e;

return oldValue;

}

public E get(int index) {

rangeCheck(index);

checkForComodification();

return ArrayList.this.elementData(offset + index);

}

public int size() {

checkForComodification();

return this.size;

}

public void add(int index, E e) {

rangeCheckForAdd(index);

checkForComodification();

parent.add(parentOffset + index, e);

this.modCount = parent.modCount;

this.size++;

}

public E remove(int index) {

rangeCheck(index);

checkForComodification();

E result = parent.remove(parentOffset + index);

this.modCount = parent.modCount;

this.size--;

return result;

}

protected void removeRange(int fromIndex, int toIndex) {

checkForComodification();

parent.removeRange(parentOffset + fromIndex,

parentOffset + toIndex);

this.modCount = parent.modCount;

this.size -= toIndex - fromIndex;

}

public boolean addAll(Collection<? extends E> c) {

return addAll(this.size, c);

}

public boolean addAll(int index, Collection<? extends E> c) {

rangeCheckForAdd(index);

int cSize = c.size();

if (cSize==0)

return false;

checkForComodification();

parent.addAll(parentOffset + index, c);

this.modCount = parent.modCount;

this.size += cSize;

return true;

}

public Iterator<E> iterator() {

return listIterator();

}

public ListIterator<E> listIterator(final int index) {

checkForComodification();

rangeCheckForAdd(index);

final int offset = this.offset;

return new ListIterator<E>() {

int cursor = index;

int lastRet = -1;

int expectedModCount = ArrayList.this.modCount;

public boolean hasNext() {

return cursor != SubList.this.size;

}

@SuppressWarnings("unchecked")

public E next() {

checkForComodification();

int i = cursor;

if (i >= SubList.this.size)

throw new NoSuchElementException();

Object[] elementData = ArrayList.this.elementData;

if (offset + i >= elementData.length)

throw new ConcurrentModificationException();

cursor = i + 1;

return (E) elementData[offset + (lastRet = i)];

}

public boolean hasPrevious() {

return cursor != 0;

}

@SuppressWarnings("unchecked")

public E previous() {

checkForComodification();

int i = cursor - 1;

if (i < 0)

throw new NoSuchElementException();

Object[] elementData = ArrayList.this.elementData;

if (offset + i >= elementData.length)

throw new ConcurrentModificationException();

cursor = i;

return (E) elementData[offset + (lastRet = i)];

}

@SuppressWarnings("unchecked")

public void forEachRemaining(Consumer<? super E> consumer) {

Objects.requireNonNull(consumer);

final int size = SubList.this.size;

int i = cursor;

if (i >= size) {

return;

}

final Object[] elementData = ArrayList.this.elementData;

if (offset + i >= elementData.length) {

throw new ConcurrentModificationException();

}

while (i != size && modCount == expectedModCount) {

consumer.accept((E) elementData[offset + (i++)]);

}

// update once at end of iteration to reduce heap write traffic

lastRet = cursor = i;

checkForComodification();

}

public int nextIndex() {

return cursor;

}

public int previousIndex() {

return cursor - 1;

}

public void remove() {

if (lastRet < 0)

throw new IllegalStateException();

checkForComodification();

try {

SubList.this.remove(lastRet);

cursor = lastRet;

lastRet = -1;

expectedModCount = ArrayList.this.modCount;

} catch (IndexOutOfBoundsException ex) {

throw new ConcurrentModificationException();

}

}

public void set(E e) {

if (lastRet < 0)

throw new IllegalStateException();

checkForComodification();

try {

ArrayList.this.set(offset + lastRet, e);

} catch (IndexOutOfBoundsException ex) {

throw new ConcurrentModificationException();

}

}

public void add(E e) {

checkForComodification();

try {

int i = cursor;

SubList.this.add(i, e);

cursor = i + 1;

lastRet = -1;

expectedModCount = ArrayList.this.modCount;

} catch (IndexOutOfBoundsException ex) {

throw new ConcurrentModificationException();

}

}

final void checkForComodification() {

if (expectedModCount != ArrayList.this.modCount)

throw new ConcurrentModificationException();

}

};

}

public List<E> subList(int fromIndex, int toIndex) {

subListRangeCheck(fromIndex, toIndex, size);

return new SubList(this, offset, fromIndex, toIndex);

}

private void rangeCheck(int index) {

if (index < 0 || index >= this.size)

throw new IndexOutOfBoundsException(outOfBoundsMsg(index));

}

private void rangeCheckForAdd(int index) {

if (index < 0 || index > this.size)

throw new IndexOutOfBoundsException(outOfBoundsMsg(index));

}

private String outOfBoundsMsg(int index) {

return "Index: "+index+", Size: "+this.size;

}

private void checkForComodification() {

if (ArrayList.this.modCount != this.modCount)

throw new ConcurrentModificationException();

}

public Spliterator<E> spliterator() {

checkForComodification();

return new ArrayListSpliterator<E>(ArrayList.this, offset,

offset + this.size, this.modCount);

}

}

@Override

public void forEach(Consumer<? super E> action) {

Objects.requireNonNull(action);

final int expectedModCount = modCount;

@SuppressWarnings("unchecked")

final E[] elementData = (E[]) this.elementData;

final int size = this.size;

for (int i=0; modCount == expectedModCount && i < size; i++) {

action.accept(elementData[i]);

}

if (modCount != expectedModCount) {

throw new ConcurrentModificationException();

}

}

/\*\*

\* Creates a <em><a href="Spliterator.html#binding">late-binding</a></em>

\* and <em>fail-fast</em> {@link Spliterator} over the elements in this

\* list.

\*

\* <p>The {@code Spliterator} reports {@link Spliterator#SIZED},

\* {@link Spliterator#SUBSIZED}, and {@link Spliterator#ORDERED}.

\* Overriding implementations should document the reporting of additional

\* characteristic values.

\*

\* @return a {@code Spliterator} over the elements in this list

\* @since 1.8

\*/

@Override

public Spliterator<E> spliterator() {

return new ArrayListSpliterator<>(this, 0, -1, 0);

}

/\*\* Index-based split-by-two, lazily initialized Spliterator \*/

static final class ArrayListSpliterator<E> implements Spliterator<E> {

/\*

\* If ArrayLists were immutable, or structurally immutable (no

\* adds, removes, etc), we could implement their spliterators

\* with Arrays.spliterator. Instead we detect as much

\* interference during traversal as practical without

\* sacrificing much performance. We rely primarily on

\* modCounts. These are not guaranteed to detect concurrency

\* violations, and are sometimes overly conservative about

\* within-thread interference, but detect enough problems to

\* be worthwhile in practice. To carry this out, we (1) lazily

\* initialize fence and expectedModCount until the latest

\* point that we need to commit to the state we are checking

\* against; thus improving precision. (This doesn't apply to

\* SubLists, that create spliterators with current non-lazy

\* values). (2) We perform only a single

\* ConcurrentModificationException check at the end of forEach

\* (the most performance-sensitive method). When using forEach

\* (as opposed to iterators), we can normally only detect

\* interference after actions, not before. Further

\* CME-triggering checks apply to all other possible

\* violations of assumptions for example null or too-small

\* elementData array given its size(), that could only have

\* occurred due to interference. This allows the inner loop

\* of forEach to run without any further checks, and

\* simplifies lambda-resolution. While this does entail a

\* number of checks, note that in the common case of

\* list.stream().forEach(a), no checks or other computation

\* occur anywhere other than inside forEach itself. The other

\* less-often-used methods cannot take advantage of most of

\* these streamlinings.

\*/

private final ArrayList<E> list;

private int index; // current index, modified on advance/split

private int fence; // -1 until used; then one past last index

private int expectedModCount; // initialized when fence set

/\*\* Create new spliterator covering the given range \*/

ArrayListSpliterator(ArrayList<E> list, int origin, int fence,

int expectedModCount) {

this.list = list; // OK if null unless traversed

this.index = origin;

this.fence = fence;

this.expectedModCount = expectedModCount;

}

private int getFence() { // initialize fence to size on first use

int hi; // (a specialized variant appears in method forEach)

ArrayList<E> lst;

if ((hi = fence) < 0) {

if ((lst = list) == null)

hi = fence = 0;

else {

expectedModCount = lst.modCount;

hi = fence = lst.size;

}

}

return hi;

}

public ArrayListSpliterator<E> trySplit() {

int hi = getFence(), lo = index, mid = (lo + hi) >>> 1;

return (lo >= mid) ? null : // divide range in half unless too small

new ArrayListSpliterator<E>(list, lo, index = mid,

expectedModCount);

}

public boolean tryAdvance(Consumer<? super E> action) {

if (action == null)

throw new NullPointerException();

int hi = getFence(), i = index;

if (i < hi) {

index = i + 1;

@SuppressWarnings("unchecked") E e = (E)list.elementData[i];

action.accept(e);

if (list.modCount != expectedModCount)

throw new ConcurrentModificationException();

return true;

}

return false;

}

public void forEachRemaining(Consumer<? super E> action) {

int i, hi, mc; // hoist accesses and checks from loop

ArrayList<E> lst; Object[] a;

if (action == null)

throw new NullPointerException();

if ((lst = list) != null && (a = lst.elementData) != null) {

if ((hi = fence) < 0) {

mc = lst.modCount;

hi = lst.size;

}

else

mc = expectedModCount;

if ((i = index) >= 0 && (index = hi) <= a.length) {

for (; i < hi; ++i) {

@SuppressWarnings("unchecked") E e = (E) a[i];

action.accept(e);

}

if (lst.modCount == mc)

return;

}

}

throw new ConcurrentModificationException();

}

public long estimateSize() {

return (long) (getFence() - index);

}

public int characteristics() {

return Spliterator.ORDERED | Spliterator.SIZED | Spliterator.SUBSIZED;

}

}

@Override

public boolean removeIf(Predicate<? super E> filter) {

Objects.requireNonNull(filter);

// figure out which elements are to be removed

// any exception thrown from the filter predicate at this stage

// will leave the collection unmodified

int removeCount = 0;

final BitSet removeSet = new BitSet(size);

final int expectedModCount = modCount;

final int size = this.size;

for (int i=0; modCount == expectedModCount && i < size; i++) {

@SuppressWarnings("unchecked")

final E element = (E) elementData[i];

if (filter.test(element)) {

removeSet.set(i);

removeCount++;

}

}

if (modCount != expectedModCount) {

throw new ConcurrentModificationException();

}

// shift surviving elements left over the spaces left by removed elements

final boolean anyToRemove = removeCount > 0;

if (anyToRemove) {

final int newSize = size - removeCount;

for (int i=0, j=0; (i < size) && (j < newSize); i++, j++) {

i = removeSet.nextClearBit(i);

elementData[j] = elementData[i];

}

for (int k=newSize; k < size; k++) {

elementData[k] = null; // Let gc do its work

}

this.size = newSize;

if (modCount != expectedModCount) {

throw new ConcurrentModificationException();

}

modCount++;

}

return anyToRemove;

}

@Override

@SuppressWarnings("unchecked")

public void replaceAll(UnaryOperator<E> operator) {

Objects.requireNonNull(operator);

final int expectedModCount = modCount;

final int size = this.size;

for (int i=0; modCount == expectedModCount && i < size; i++) {

elementData[i] = operator.apply((E) elementData[i]);

}

if (modCount != expectedModCount) {

throw new ConcurrentModificationException();

}

modCount++;

}

@Override

@SuppressWarnings("unchecked")

public void sort(Comparator<? super E> c) {

final int expectedModCount = modCount;

Arrays.sort((E[]) elementData, 0, size, c);

if (modCount != expectedModCount) {

throw new ConcurrentModificationException();

}

modCount++;

}

}