Continue

Java methods list

List Interface Explanation A List is an ordered collection that allows elements to be accessed by their integer index and searched. It can contain duplicate elements, unlike sets. This interface provides additional stipulations on iterator methods, positional access, and search operations. Key features include the ability to insert and replace elements, bidirectional access, and efficient insertion and removal of multiple elements. However, some implementations may have restrictions on the types of elements they contain, and certain operations may be performed differently depending on the element's eligibility. interface is a member of the Java Collections Framework since version 1.2 See Also: Collection, Set, ArrayList, LinkedList, Vector, Arrays.asList(Object[]), Collections.nCopies(int, Object), Collections.EMPTY_LIST, AbstractList, AbstractSequentialList parallelStream, removeIf, stream int size() Returns the number of elements in this list. If this list contains more than Integer.MAX_VALUE elements, returns Integer.MAX_VALUE. Specified by: size in interface Collection Returns: the number of elements in this list boolean isEmpty() Returns true if this list contains no elements. Specified by: isEmpty in interface Collection Returns: true if this list contains no elements boolean contains(Object o) Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)). Specified by: contains in interface Collection Parameters: o - element whose presence in this list is to be tested Returns: true if this list contains the specified element Throws: ClassCastException - if the type of the specified element is incompatible with this list (optional) NullPointerException - if the specified element is null and this list does not permit null elements (optional) Iterator iterator() Returns an iterator over the elements in this list in proper sequence. Specified by: iterator in interface Collection Specified by: iterator in interface Iterable Returns: an iterator over the elements in this list in proper sequence Object[] toArray() Returns an array containing all of the elements in this list in proper sequence (from first to last element). 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 even if this list is backed by an array). The caller is thus free to modify the returned array. This method acts as bridge between array-based and collection-based APIs. Specified by: toArray in interface Collection Returns: an array containing all of the elements in this list in proper sequence See Also: Arrays.asList(Object[]) T[] toArray(T[] a) 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. 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 list is set to null. (This is useful in determining the length of the list only if the caller knows that the list does not contain any null elements.) Like the toArray() method, this method acts as bridge between array-based and collection-based APIs. Further, this method allows precise control over the runtime type of the output array, and may, under certain circumstances, be used to save allocation costs. Suppose x is a list known to contain only strings. The following code can be used to dump the list into a newly allocated array of String: String[] y = x.toArray(new String[0]); Note that toArray(new Object[0]) is identical in function to Specified by: toArray interface Collection Type Parameters: T - the runtime type of the array contains the list Parameters: a - the array into which the elements are stored Returns: an array containing the elements Throws: ArrayStoreException - if the runtime type is not supertype every element List Appends the specified element to the end (optional operation) Lists place limitations on what may be added In particular, some will refuse null and others impose restrictions Specified by: add Parameters: e - element Returns: true Throws: UnsupportedOperationException - if not supported ClassCastException - if the class prevents it NullPointerException - if the element is null IllegalArgumentException - if the element prevents it Removes the first occurrence of the specified element from this list (optional operation) More formally, removes the element with the lowest index such that Returns: true if contained Throws: ClassCastException - if incompatible NullPointerException - if null UnsupportedOperationException - if not supported Returns true if contains all of the elements of the specified collection Specified by: containsAll Parameters: c - collection to be checked Returns: true if contains all Throws: ClassCastException - if one or more elements are incompatible NullPointerException - if one or more null elements and does not permit null elements Appends all of the elements in the specified collection to the end, in the order they are returned (optional) The behavior of this operation is undefined if the specified collection is modified while the operation is in progress. If the specified collection is this list and it's nonempty, this will occur. This is specified by the addAll method in interface Collection. The replaceAll method in a List interface allows you to replace each element with the result of applying a given operator to that element, throwing UnsupportedOperationException if this list is unmodifiable. #### Methods and Implementations - The default implementation uses a ListIterator to iterate over the elements of the list, applying the specified operator to each element using the apply method. This process continues until all elements have been processed. - If any element cannot be replaced or modification is not supported in general, an UnsupportedOperationException may be thrown. #### Requirements and Restrictions - The replaceAll operation requires that the specified operator is not null and does not return a null value if this list does not permit null elements. - The list must implement the Comparable interface for all elements to be compared using a given comparator. - Modifiable lists are required, but they do not need to be resizable. It utilizes methods borrowed from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity" in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, January 1993, pp. 467-474. Parameters: c - comparator used to compare list elements; null value implies using natural ordering Throws: ClassCastException if list contains non-comparable elements using specified comparator, UnsupportedOperationException if list's iterator does not support set operation, IllegalArgumentException if comparator violates Comparator contract Since: 1.8 void clear() Removes all list elements (optional operation). List will be empty after this call returns Specified by: clear in interface Collection Throws: UnsupportedOperationException if clear operation is not supported boolean equals(Object o) Compares specified object with this list for equality Returns true if object is also a list, both have same size, and corresponding pairs of elements are equal (Two elements e1 and e2 are equal if (e1==null ? e2==null : e1.equals(e2))). Two lists are equal if they contain same elements in same order. This ensures equals method works properly across different List interface implementations Specified by: equals in interface Collection Overrides: equals in class Object Parameters: o - object to be compared for equality with this list Returns: true if specified object is equal to this list See Also: Object.hashCode(), HashMap int hashCode() Returns hash code value for this list. Hash code of a list is defined as result of the following calculation: int hashCode = 1; for (E e : list) hashCode = 31*hashCode + (e==null ? 0 : e.hashCode()); This ensures that list1.equals(list2) implies that list1.hashCode()==list2.hashCode() for any two lists, list1 and list2, as required by general contract of Object.hashCode(). Specified by: hashCode in interface Collection Overrides: hashCode in class Object Returns: hash code value for this list See Also: Object.equals(Object), equals(Object) E get(int index) Returns element at specified position in this list Parameters: index - index of the element to return Returns: element at specified position in this list Throws: IndexOutOfBoundsException if index is out of range (index < 0 || index >= size()) E set(int index, E element) Replaces element at specified position in this list with specified element (optional operation). Parameters: index - index of the element to replace element - element to be stored at the specified position Returns: element previously at the specified position Throws: UnsupportedOperationException if set operation is not supported by this list ClassCastException if class of specified element prevents it from being added to this list NullPointerException if specified element is null and this list does not permit null elements IllegalArgumentException if some property of specified element prevents it from being added to this list IndexOutOfBoundsException if index is out of range void add(int index, E element) If the specified index is out of range, throw IndexOutOfBoundsException. Otherwise, inserts the specified element at the specified position in this list and shifts elements after it. Throws: UnsupportedOperationException - if operation is not supported by this list ClassCastException - if the class of the specified element prevents it from being added to this list NullPointerException - if the specified element is null and this list does not permit null elements IllegalArgumentException - if some property of the specified element prevents it from being added to this list E remove(int index) If the specified index is out of range, throw IndexOutOfBoundsException. Otherwise, removes the element at the specified position in this list and shifts elements before it. Returns the element that was removed from the list. Throws: UnsupportedOperationException - if operation is not supported by this list IndexOutOfBoundsException - if the index is out of range (index < 0 || index > size()) int indexOf(Object o) If the specified element appears in this list, returns its index. Otherwise, returns -1. Throws: ClassCastException - if the type of the specified element is incompatible with this list NullPointerException - if the specified element is null and this list does not permit null elements int lastIndexOf(Object o) If the specified element appears in this list, returns its most recent index. Otherwise, returns -1. Throws: ClassCastException - if the type of the specified element is incompatible with this list NullPointerException - if the specified element is null and this list does not permit null elements ListIterator listIterator() Returns a list iterator over the elements in this list. Throws: UnsupportedOperationException - if operation is not supported by this list ListIterator listIterator(int index) If the specified index is out of range, throw IndexOutOfBoundsException. Otherwise, returns a list iterator starting at the specified position. Returns an iterator over the list's elements, starting from a specified position. This method throws IndexOutOfBoundsException if the index is out of range or not provided. ```java List subList(int fromIndex, int toIndex) ``` * Returns a view of the portion of this list between the specified indices (fromIndex inclusive and toIndex exclusive). * The returned list supports all optional operations. * Structural modifications on the backing list invalidate the semantics of the returned list. The provided documentation is a mix of Java API references, class implementations, and usage guidelines for the List interface. ArrayList and LinkedList are two classes in Java that implement the List interface, allowing them to use common methods for adding, accessing, updating, and removing elements from a collection. Unlike fixed-size arrays, ArrayLists are more dynamic, enabling the addition of items as needed. To create an ArrayList, import the ArrayList class and define a new object using angle brackets to specify data types. For example: ```java import java.util.ArrayList; public class Main { public static void main(String[] args) { ArrayList students = new ArrayList(); } } ``` In this code snippet, the element "John" is added to the ArrayList using the `add()` method. To access elements in an ArrayList, use the `get()` method with the desired index. For instance: ```java import java.util.ArrayList; public class Main { public static void main(String[] args) { ArrayList students = new ArrayList(); students.add("John"); System.out.println(students.get(1)); } } ``` In this code, element "Jane" is accessed using the `get()` method with index 1. To update the value of an element in an ArrayList, use the `set()` method with the desired index and new value. For example: ```java import java.util.ArrayList; public class Main { public static void main(String[] args) { ArrayList students = new ArrayList(); students.add("John"); students.add("Jane"); students.set(2, "Jade"); System.out.println(students); } } ``` In this code, the value of element at index 2 is updated to "Jade". The `remove()` method can be used to remove elements from an ArrayList by specifying their indices. For instance: ```java import java.util.ArrayList; public class Main { public static void main(String[] args) { ArrayList students = new ArrayList(); students.add("John"); students.add("Jane"); students.remove(2); System.out.println(students); } } ``` In this code, element "Doe" is removed using the `remove()` method with index 2. The `clear()` method can be used to remove all elements from an ArrayList. For example: ```java import java.util.ArrayList; public class Main { public static void main(String[] args) { ArrayList students = new ArrayList(); students.add("John"); students.add("Jane"); students.clear(); System.out.println(students); } } ``` In this code, all elements from the ArrayList are cleared using the `clear()` method. It's worth noting that LinkedList has additional methods like `addFirst()`, `addLast()`, and `getFirst()`, but these are not part of the standard List interface. The List interface is a fundamental concept in Java used for storing ordered collections. This article has reviewed its capabilities by exploring some key operations such as getting and removing elements from the list. We also examined the implementation classes of the List interface, particularly ArrayList and LinkedList, which are commonly employed due to their efficiency. Through code examples, we demonstrated how to manipulate a collection using these classes, highlighting both shared methods and the unique features of the LinkedList class.