CS112
Lists
Chapters 20

Lecture 15

الفصل الدراسي الثاني 1443- 2022 Spring

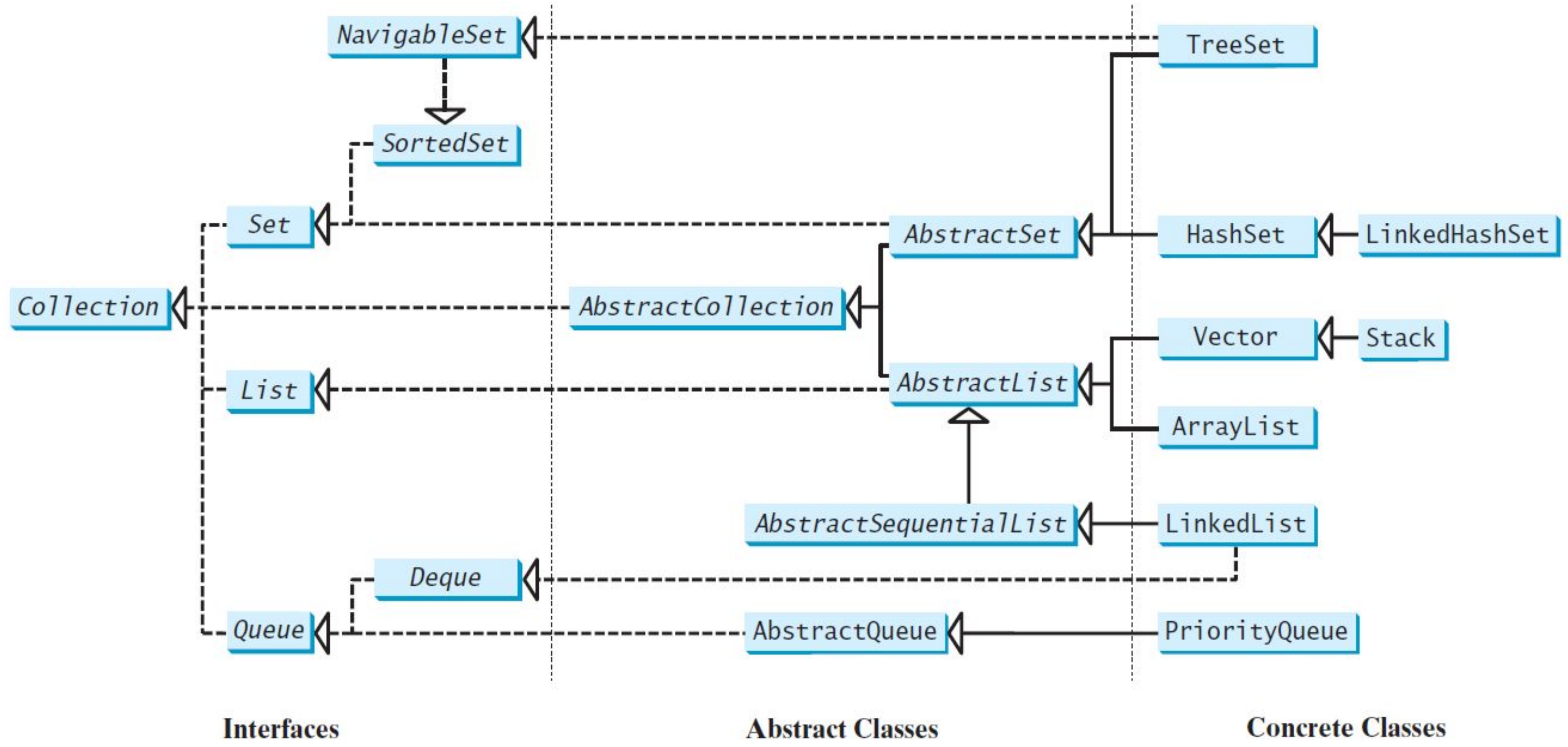
College of Computer Science and Engineering

Java Collection Framework Hierarchy (1/2)

- A *collection* is a container object that holds a group of objects, often referred to as *elements*.
- The Java Collections Framework supports three types of collections:
    - *Lists*
    - *Sets*
    - *Maps*.

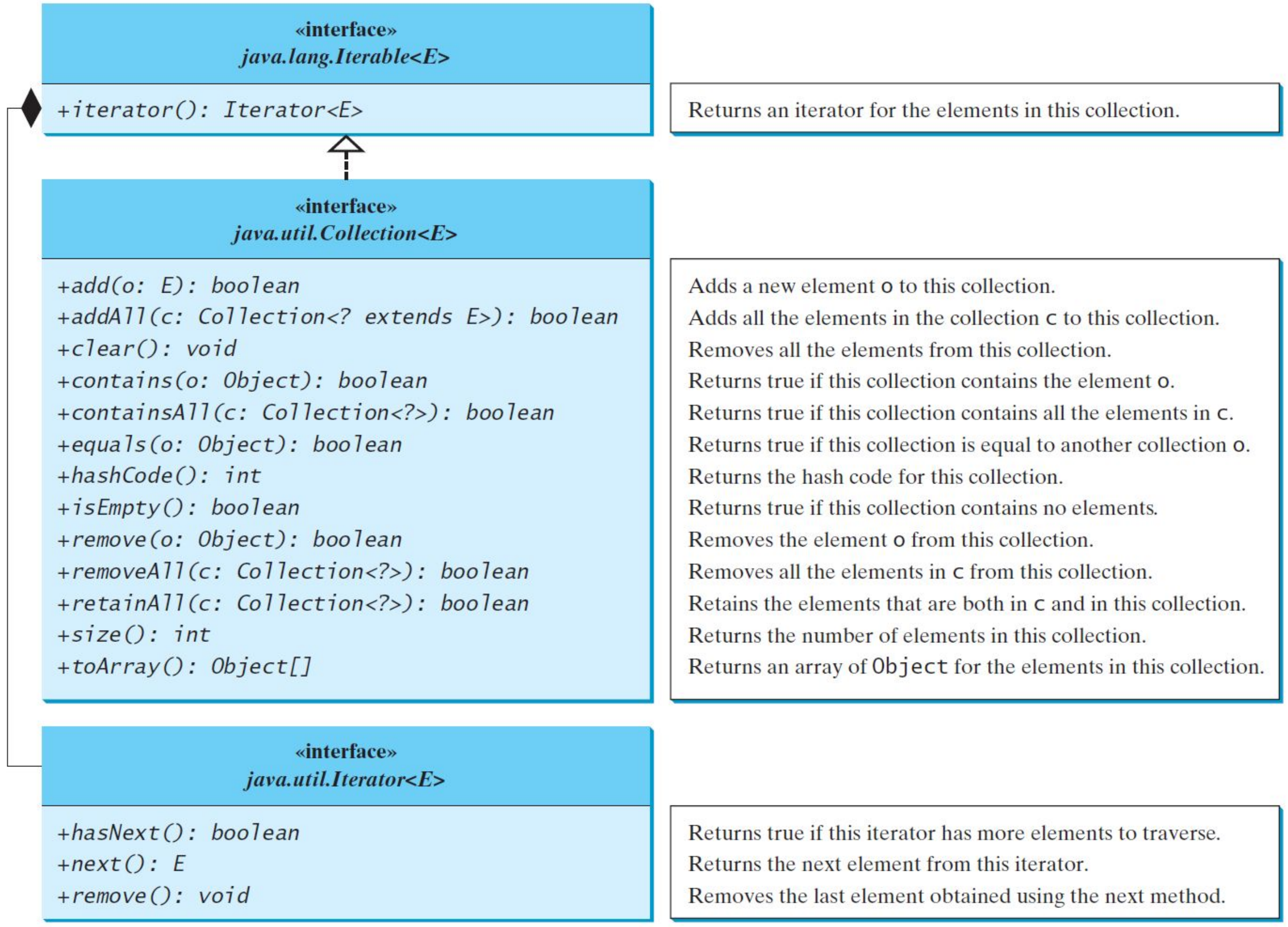
Java Collection Framework Hierarchy (2/2)

- Set and List are subinterfaces of Collection.



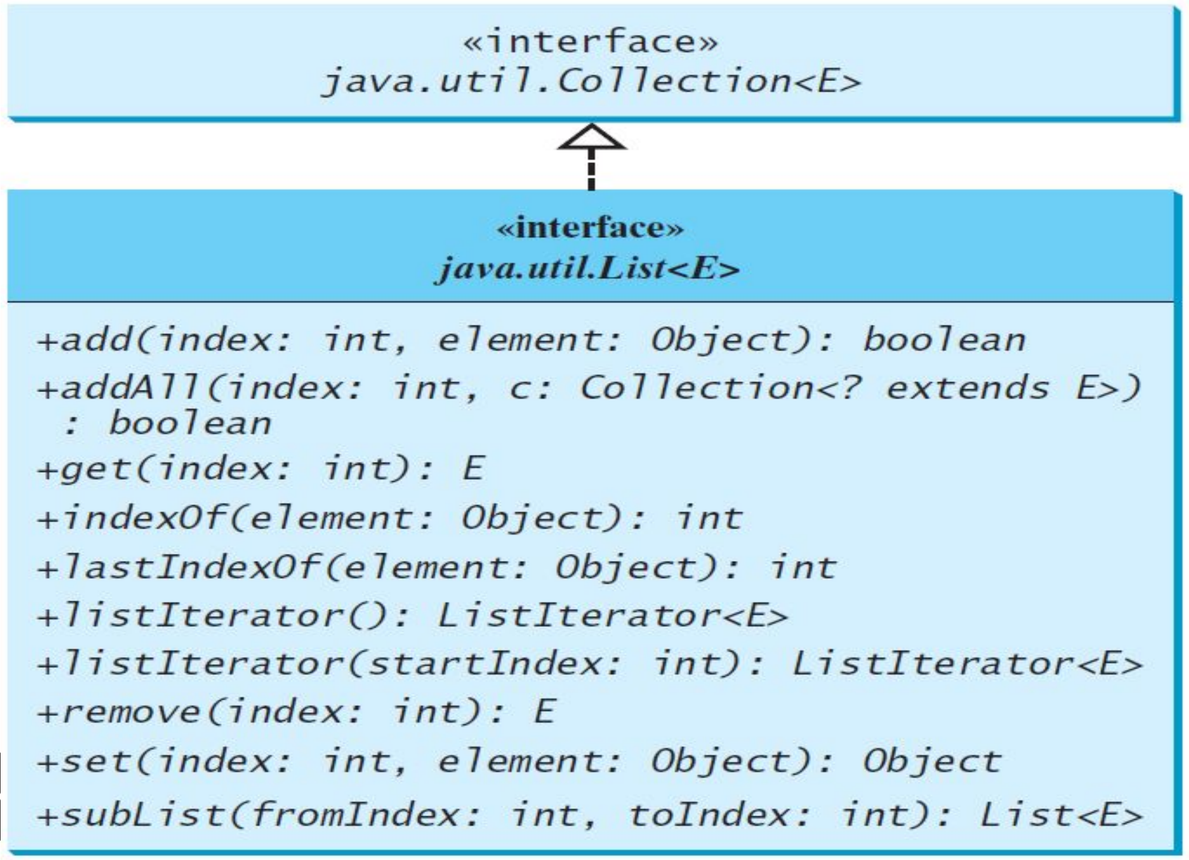
The Collection Interface

- The Collection interface is the root interface for manipulating a collection of objects.
- See TestCollection.java: Page 764


«interface»
java.lang.Iterable<E>
+iterator(): Iterator<E>
Returns an iterator for the elements in this collection.
«interface»
java.util.Collection<E>
+add(o: E): boolean
+addAll(c: Collection<? extends E>): boolean
+clear(): void
+contains(o: Object): boolean
+containsAll(c: Collection<?>): boolean
+equals(o: Object): boolean
+hashCode(): int
+isEmpty(): boolean
+remove(o: Object): boolean
+removeAll(c: Collection<?>): boolean
+retainAll(c: Collection<?>): boolean
+size(): int
+toArray(): Object[]
Adds a new element o to this collection.
Adds all the elements in the collection c to this collection.
Removes all the elements from this collection.
Returns true if this collection contains the element o.
Returns true if this collection contains all the elements in c.
Returns true if this collection is equal to another collection o.
Returns the hash code for this collection.
Returns true if this collection contains no elements.
Removes the element o from this collection.
Removes all the elements in c from this collection.
Retains the elements that are both in c and in this collection.
Returns the number of elements in this collection.
Returns an array of Object for the elements in this collection.
«interface»
java.util.Iterator<E>
+hasNext(): boolean
+next(): E
+remove(): void
Returns true if this iterator has more elements to traverse.
Returns the next element from this iterator.
Removes the last element obtained using the next method.

The List Interface

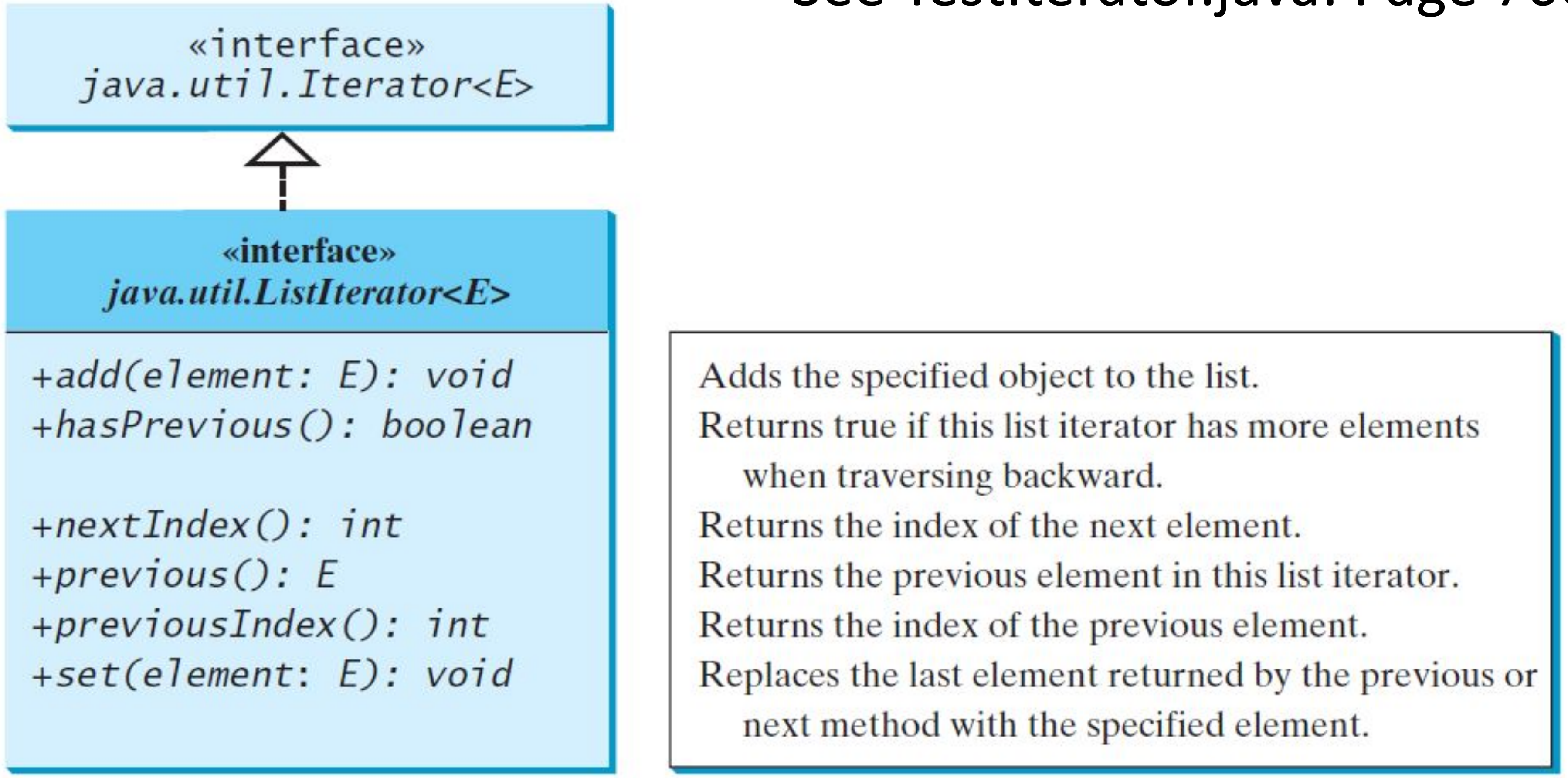
- A list stores elements in a sequential order, and allows the user to specify where the element is stored.
- The user can access the elements by index.



`+add(index: int, element: Object): boolean`	Adds a new element at the specified index.
`+addAll(index: int, c: Collection<? extends E>) : boolean`	Adds all the elements in `c` to this list at the specified index.
`+get(index: int): E`	Returns the element in this list at the specified index.
`+indexOf(element: Object): int`	Returns the index of the first matching element.
`+lastIndexOf(element: Object): int`	Returns the index of the last matching element.
`+listIterator(): ListIterator<E>`	Returns the list iterator for the elements in this list.
`+listIterator(startIndex: int): ListIterator<E>`	Returns the iterator for the elements from `startIndex`.
`+remove(index: int): E`	Removes the element at the specified index.
`+set(index: int, element: Object): Object`	Sets the element at the specified index.
`+subList(fromIndex: int, toIndex: int): List<E>`	Returns a sublist from `fromIndex` to `toIndex-1`.

The List Iterator

- See TestIterator.java: Page 766

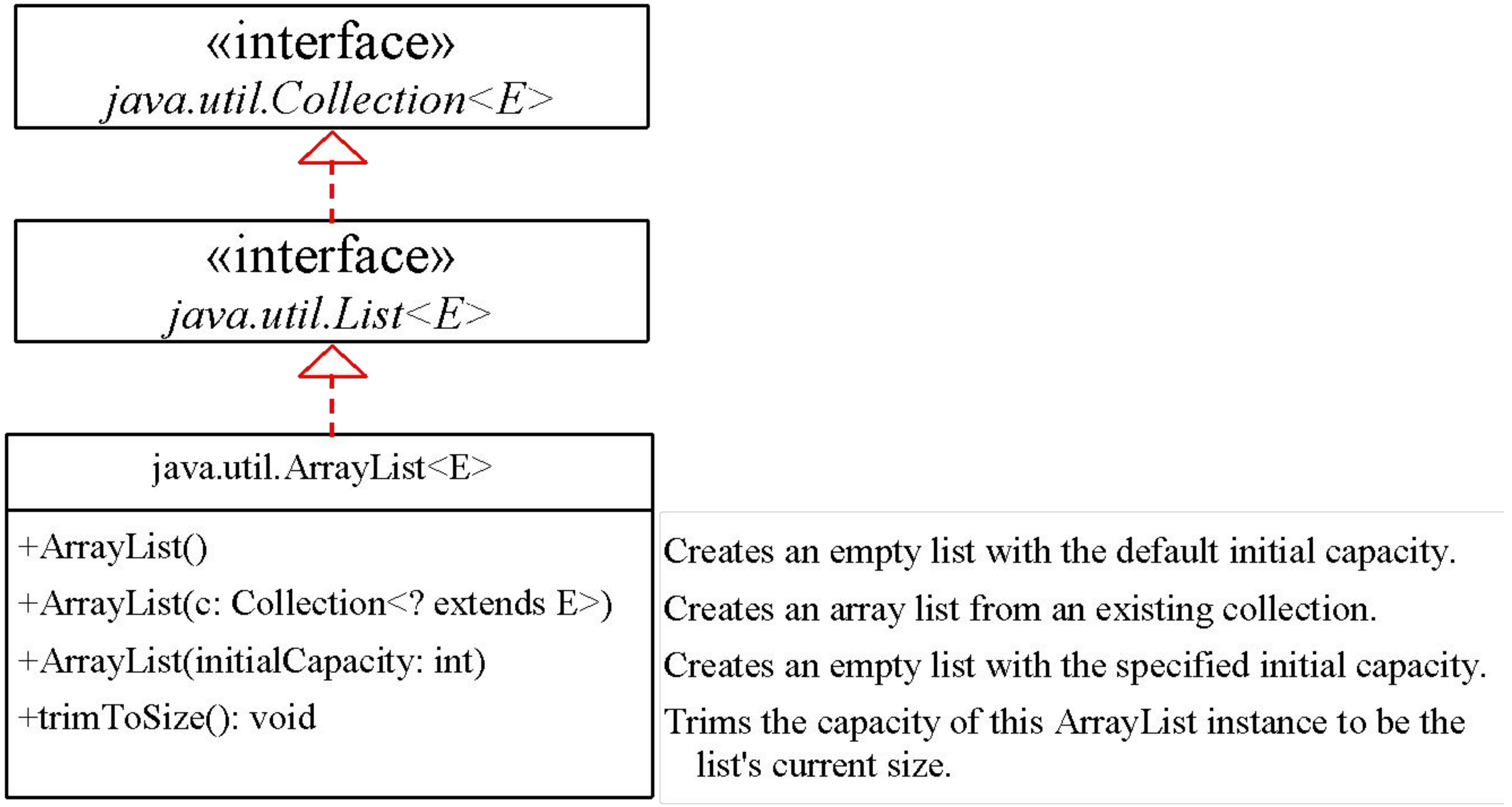


Adds the specified object to the list.
Returns true if this list iterator has more elements when traversing backward.
Returns the index of the next element.
Returns the previous element in this list iterator.
Returns the index of the previous element.
Replaces the last element returned by the previous or next method with the specified element.

ArrayList and LinkedList

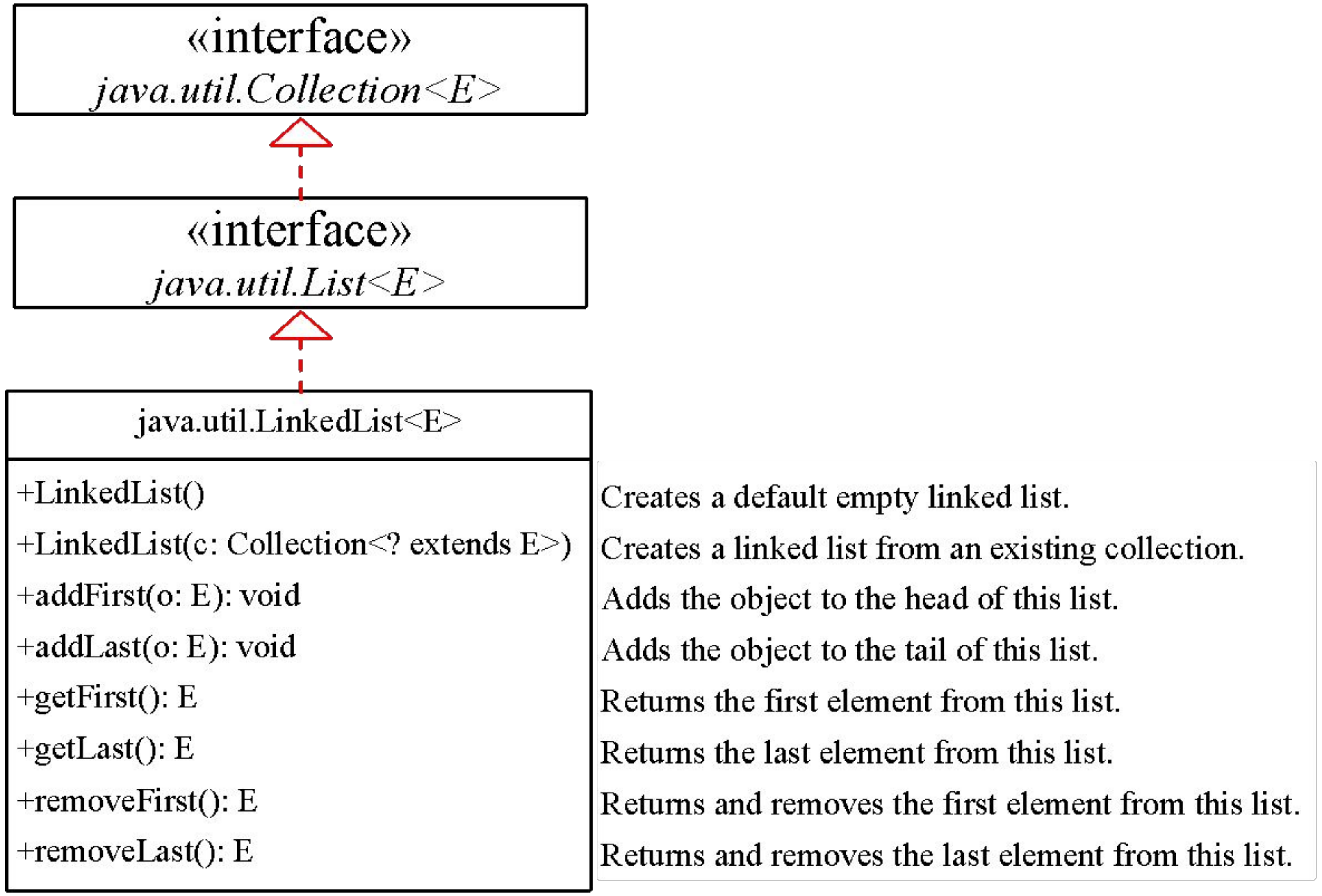
- The ArrayList class and the LinkedList class are concrete implementations of the List interface.
- Which of the two classes you use depends on your specific needs:
  - If you need to support random access through an index without inserting or removing elements from any place other than the end, ArrayList offers the most efficient collection.
  - If, however, your application requires the insertion or deletion of elements from any place in the list, you should choose LinkedList.
- A list can grow or shrink dynamically. An array is fixed once it is created. If your application does not require insertion or deletion of elements, the most efficient data structure is the array.

java.util.ArrayList



Creates an empty list with the default initial capacity.
Creates an array list from an existing collection.
Creates an empty list with the specified initial capacity.
Trims the capacity of this ArrayList instance to be the list's current size.

java.util.LinkedList



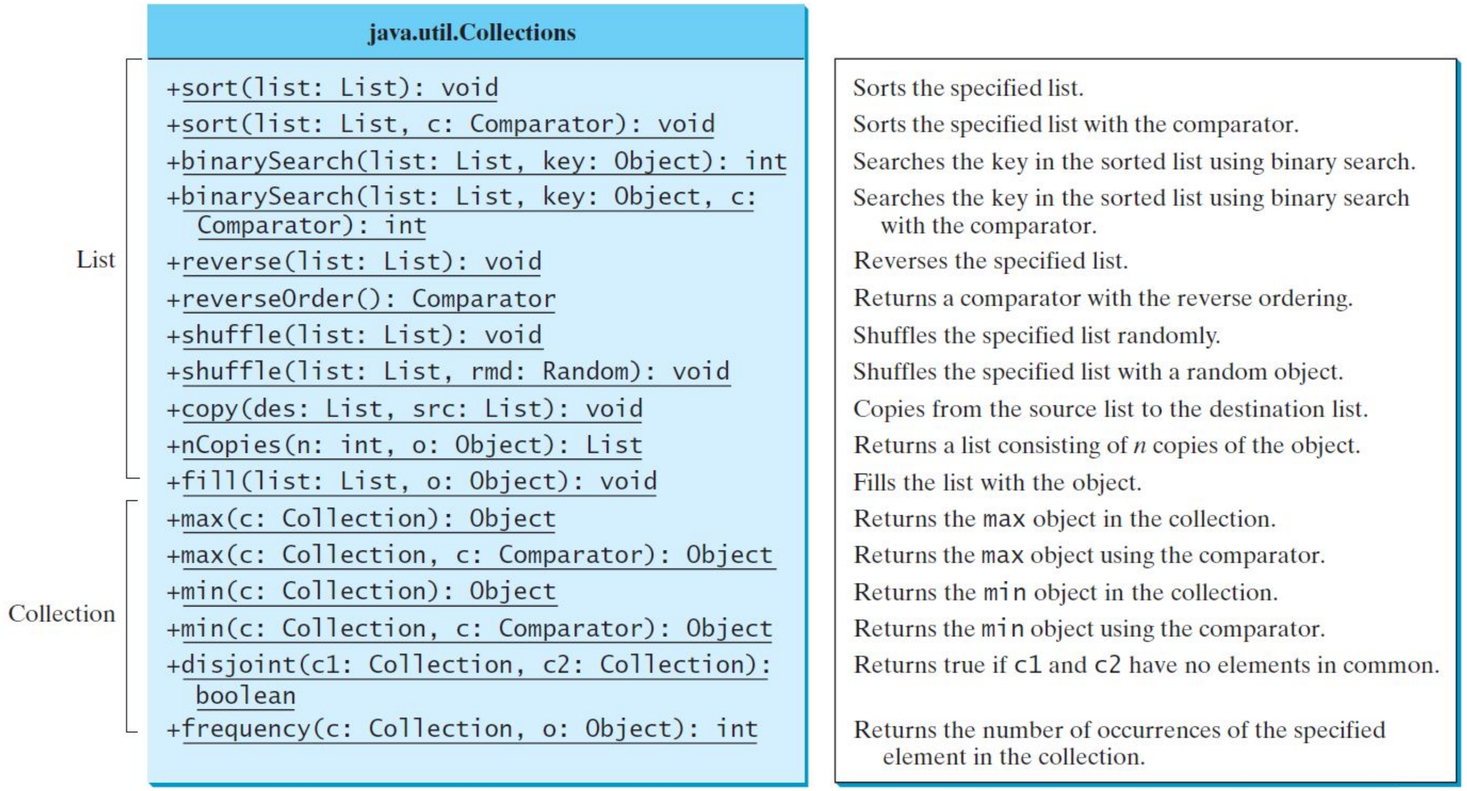
+LinkedList()	Creates a default empty linked list.
+LinkedList(c: Collection<? extends E>)	Creates a linked list from an existing collection.
+addFirst(o: E): void	Adds the object to the head of this list.
+addLast(o: E): void	Adds the object to the tail of this list.
+getFirst(): E	Returns the first element from this list.
+getLast(): E	Returns the last element from this list.
+removeFirst(): E	Returns and removes the first element from this list.
+removeLast(): E	Returns and removes the last element from this list.

Example: Using ArrayList and LinkedList

- This example creates an array list filled with numbers, and inserts new elements into the specified location in the list. The example also creates a linked list from the array list, inserts and removes the elements from the list. Finally, the example traverses the list forward and backward.
- See TestArrayAndLinkedList.java

The Collections Class

- The Collections class contains various static methods for operating on collections and maps, for creating synchronized collection classes, and for creating read-only collection classes.



	java.util.Collections	
List	+sort(list: List): void	Sorts the specified list.
	+sort(list: List, c: Comparator): void	Sorts the specified list with the comparator.
	+binarySearch(list: List, key: Object): int	Searches the key in the sorted list using binary search.
	+binarySearch(list: List, key: Object, c: Comparator): int	Searches the key in the sorted list using binary search with the comparator.
	+reverse(list: List): void	Reverses the specified list.
	+reverseOrder(): Comparator	Returns a comparator with the reverse ordering.
	+shuffle(list: List): void	Shuffles the specified list randomly.
	+shuffle(list: List, rmd: Random): void	Shuffles the specified list with a random object.
	+copy(des: List, src: List): void	Copies from the source list to the destination list.
	+nCopies(n: int, o: Object): List	Returns a list consisting of *n* copies of the object.
	+fill(list: List, o: Object): void	Fills the list with the object.
Collection	+max(c: Collection): Object	Returns the max object in the collection.
	+max(c: Collection, c: Comparator): Object	Returns the max object using the comparator.
	+min(c: Collection): Object	Returns the min object in the collection.
	+min(c: Collection, c: Comparator): Object	Returns the min object using the comparator.
	+disjoint(c1: Collection, c2: Collection): boolean	Returns true if c1 and c2 have no elements in common.
	+frequency(c: Collection, o: Object): int	Returns the number of occurrences of the specified element in the collection.

Sorting List Elements

- You can sort the comparable elements in a list in its natural order with the **compareTo** method in the **Comparable** interface.
- You may also specify a comparator to sort elements.
- For example, the following code sorts strings in a list:

```
List<String> list = Arrays.asList("red", "green", "blue");
Collections.sort(list);
System.out.println(list);
```

The output is: **[blue, green, red]**.

Search for an Element/key in a List

- You can use the **binarySearch** method to search for a key in a list. To use this method, the list must be sorted in increasing order. If the key is not in the list, the method returns -(*insertion point* +1).
- Recall that the insertion point is where the item would fall in the list if it were present.
- For example, the following code searches the keys in a list of integers and a list of strings:

```
List<Integer> list1 =
Arrays.asList(2, 4, 7, 10, 11, 45, 50, 59, 60, 66);
System.out.println("(1) Index: " + Collections.binarySearch(list1, 7));

System.out.println("(2) Index: " + Collections.binarySearch(list1, 9));
List<String> list2 = Arrays.asList("blue", "green", "red");
System.out.println("(3) Index: " +
Collections.binarySearch(list2, "red"));
System.out.println("(4) Index: " +
Collections.binarySearch(list2, "cyan"));
```

The output of the preceding code is:

```
(1) Index: 2
(2) Index: -4
(3) Index: 2
(4) Index: -2
```

Reverse Elements in a List

- You can use the **reverse** method to reverse the elements in a list.
- For example, the following code displays **[blue, green, red, yellow]:**

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");
Collections.reverse(list);
System.out.println(list);
```

Shuffle List Elements

- You can use the **shuffle(List)** method to randomly reorder the elements in a list.
- For example, the following code shuffles the elements in **list:**

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");
Collections.shuffle(list);
System.out.println(list);
```

Copy Lists

- You will see that **list1** and **list2** have the same sequence of elements before and after the shuffling.
- You can use the **copy(det, src)** method to copy all the elements from a source list to a destination list on the same index. The destination list must be as long as the source list. If it is longer, the remaining elements in the source list are not affected.
- For example, the following code copies **list2** to **list1:**

```
List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");
List<String> list2 = Arrays.asList("white", "black");
Collections.copy(list1, list2);
System.out.println(list1);
```

The output for **list1** is **[white, black, green, blue]**. The **copy** method performs a shallow copy: only the references of the elements from the source list are copied.