CISC 3115 TY3

C24a: Lists

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Outline

- Concept of data structure
- Use data structures
  - List
  - Stack
  - Queue and priority queue
  - Set and map

Outline of This Lecture

- Data structure and Java Collections
  - Concept of data structure and Java Collection Framework
  - Type hierarchy of Java Collection Framework
  - The Collection interface
- List, ArrayList, and LinkedList

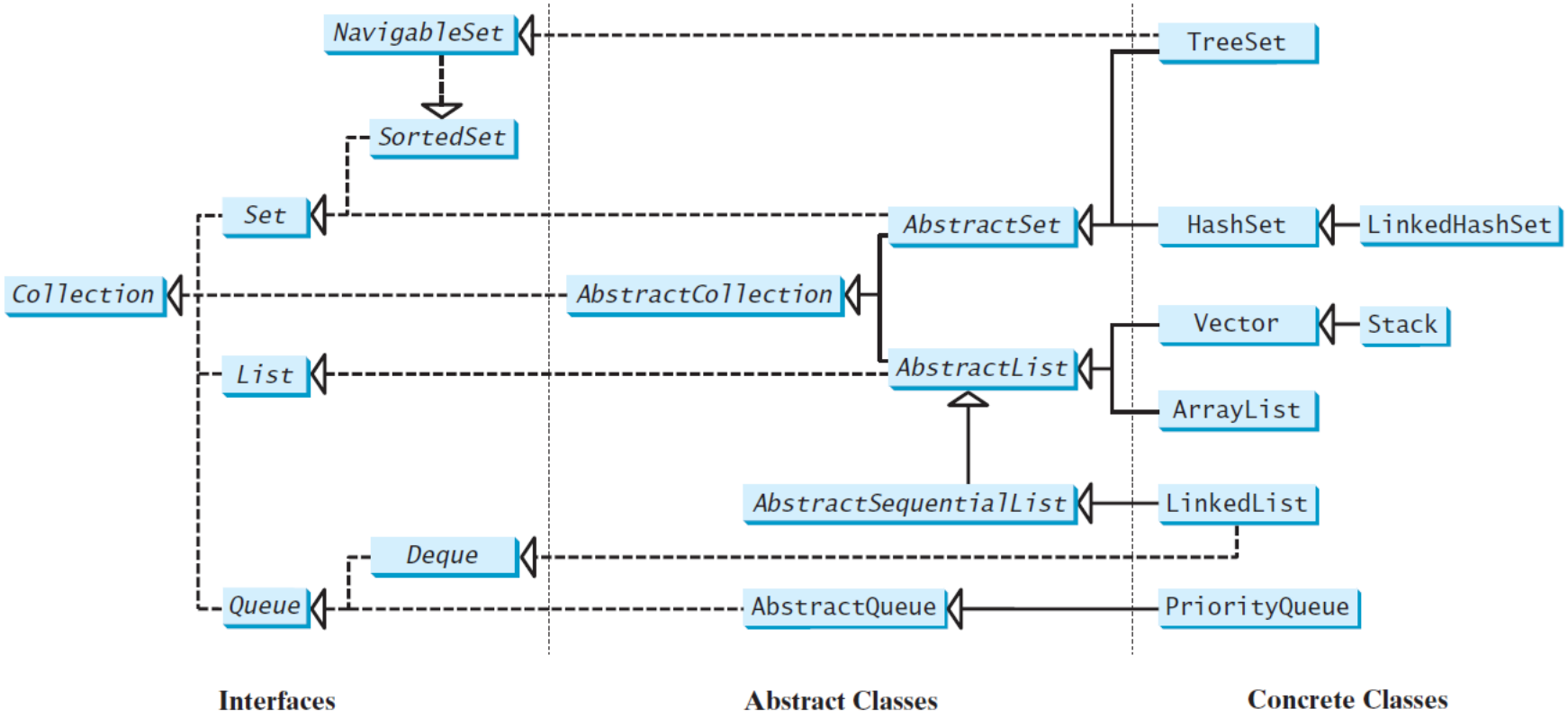
Data Structure

- A collection of data organized in some fashion in a program.
  - Data elements
  - Operations for accessing and manipulating the data elements
- In Java, how do we represent data and operations?
  - Data: primitive data type variables; objects and reference variables
  - Operations: methods

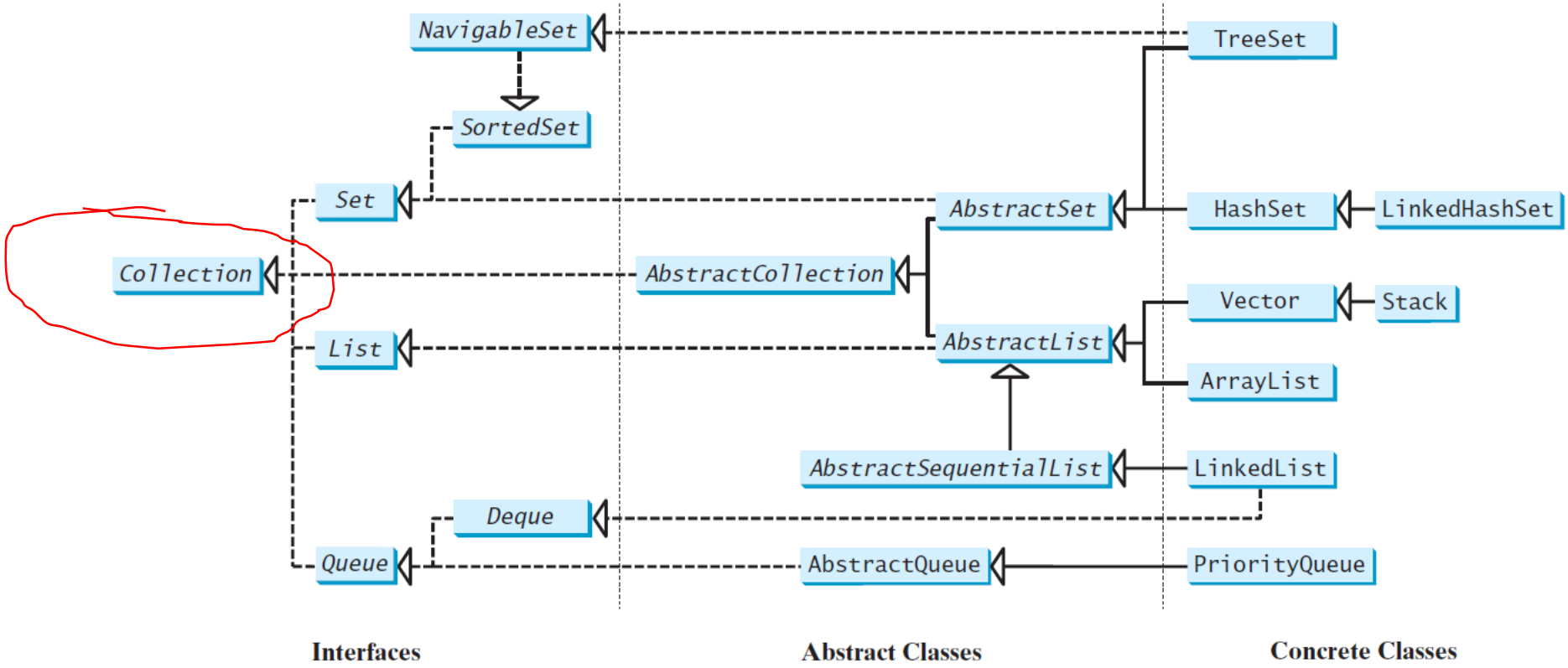
Java Collection Framework

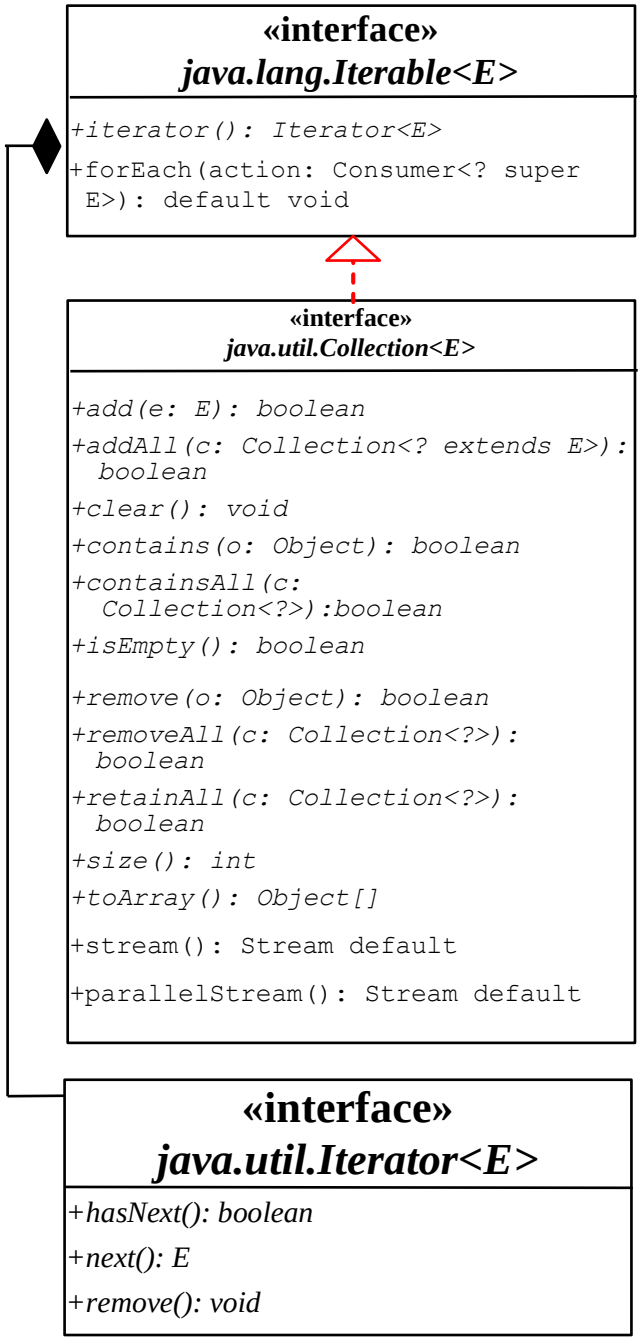
- *A collection* is an object that represents a group of objects
  - Essentially, a collection is a representation/an implementation of a data structure
- Java Collection Framework
  - A unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.
  - Data structures: list, stack, and queue

Java Collection Framework Hierarchy



The Collection Interface





Returns an iterator for the elements in this collection.

Performs an action for each element in this iterator.

Adds a new element `e` 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 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.

Returns a stream from this collection (covered in Ch 23).

Returns a parallel stream from this collection (covered in Ch 23).

Returns true if this iterator has more elements to traverse.

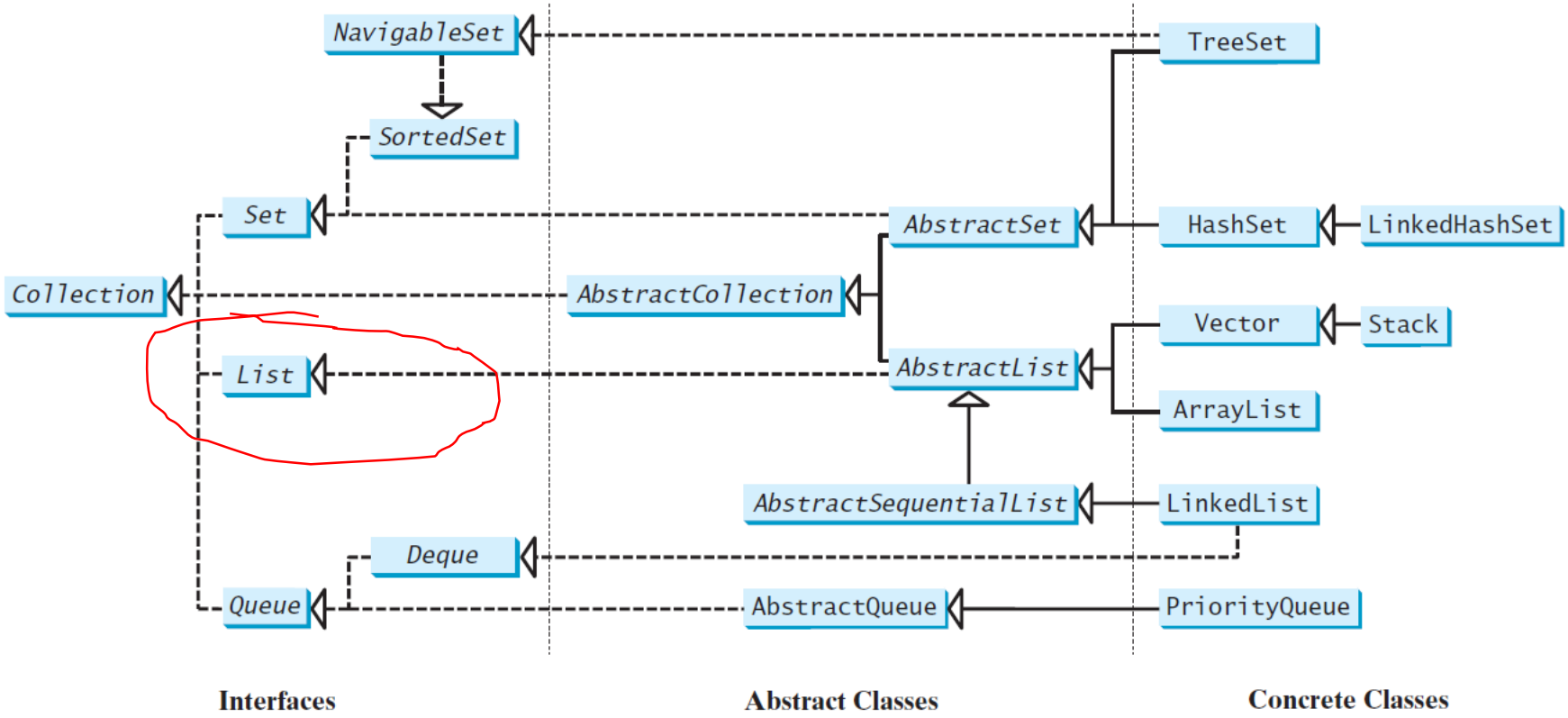
Returns the next element from this iterator.

Removes the last element obtained using the next method.

Questions?

- How about we take a look at Java API documentation about these?
- Concept of data structure
- Concept of Java Collection Framework
- Relationship between data structure and Java Collection
- Type Hierarchy of Java Collection Framework

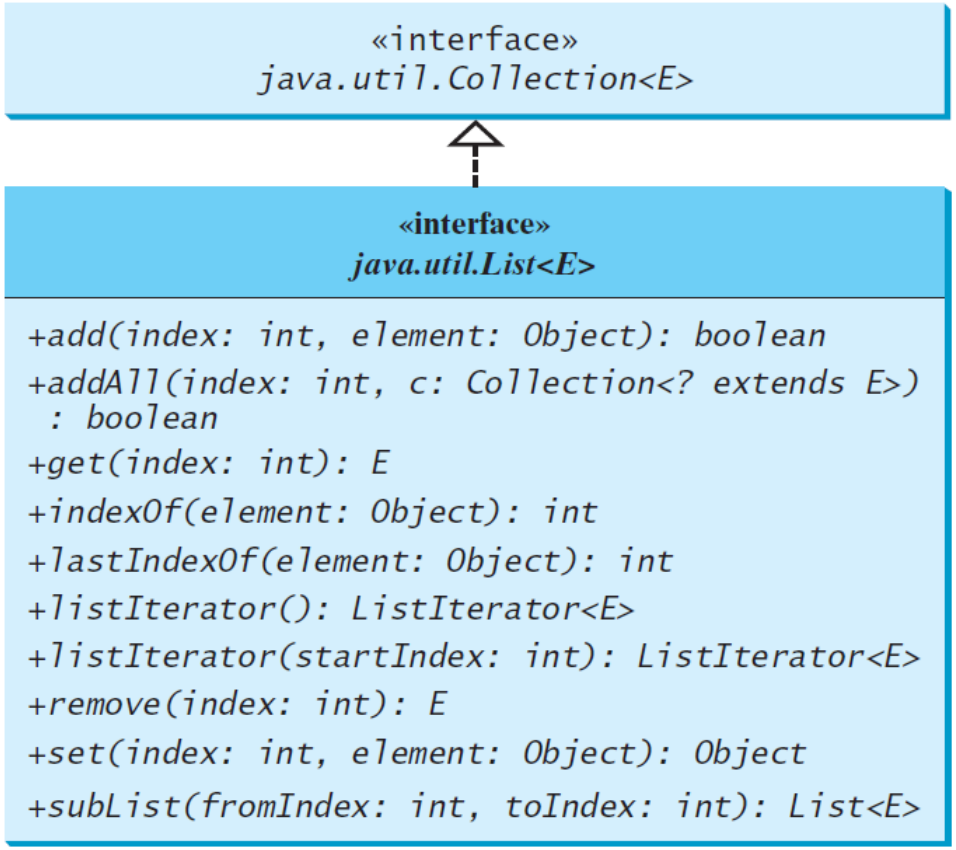
The List Interface



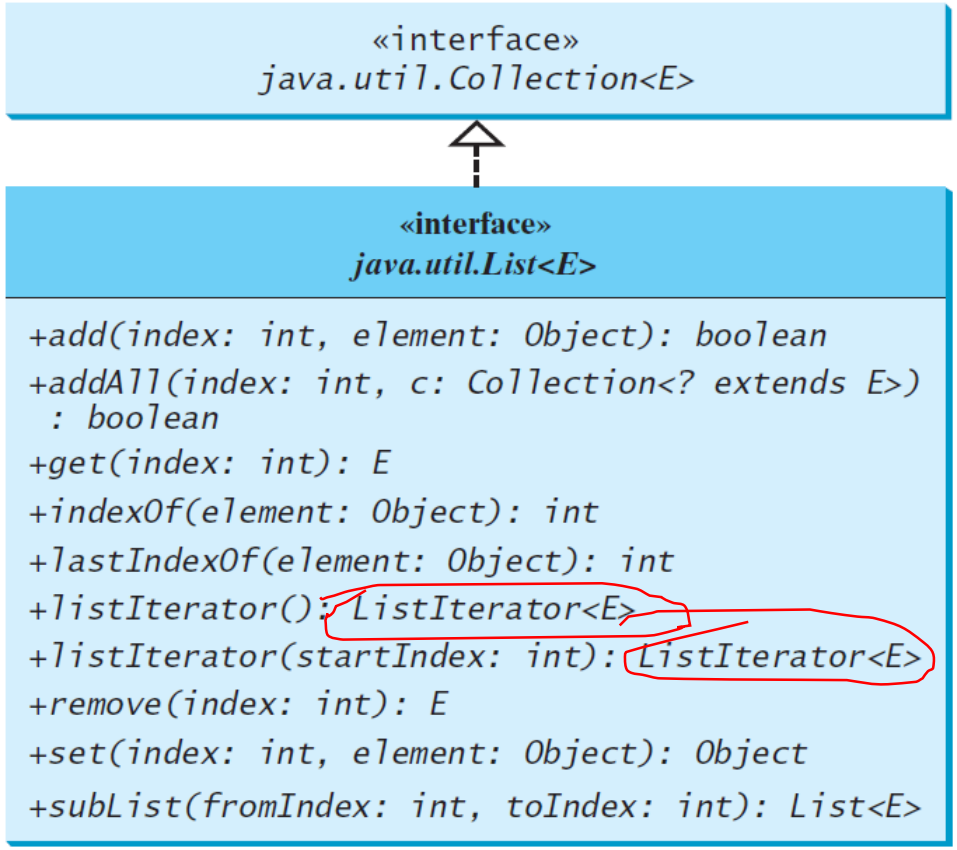
The List Data Structure

- A list stores elements in a sequential order, and allows the user to specify where the element is stored.
- The user may be able to access the elements by index.
  - However, in general, one should not assume that it takes equal amount of time to access different elements using their indices

The List Interface

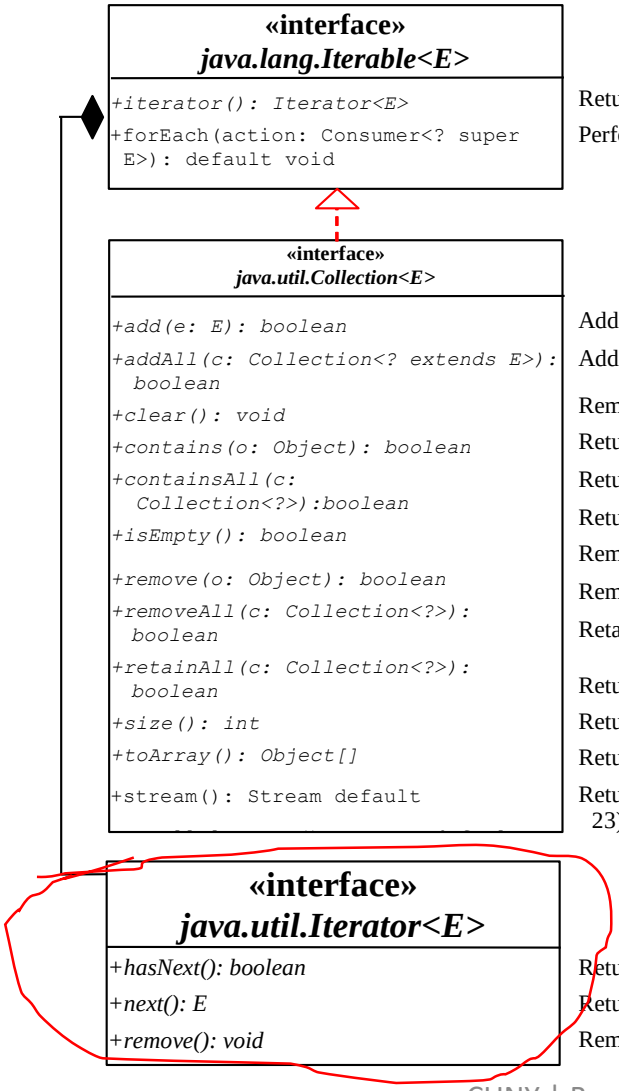


ListIterator?



`+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`.

Recall: The Collection Interface



Returns an iterator for the elements in this collection.

Performs an action for each element in this iterator.

Adds a new element e 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 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.

Returns a stream from this collection (covered in Ch 23).

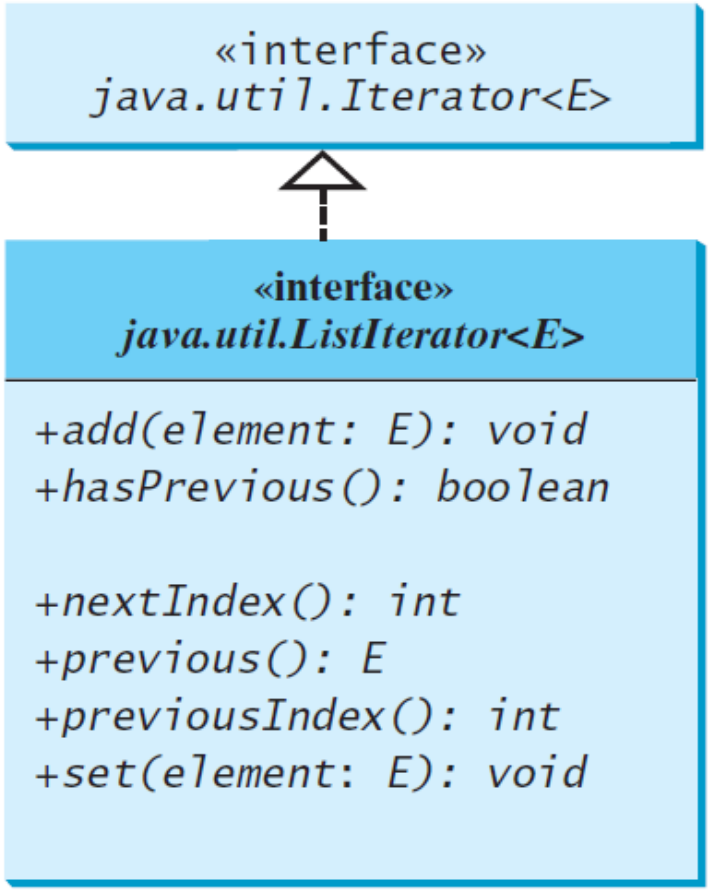
Returns a parallel stream from this collection (covered in Ch 23).

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 ListIterator

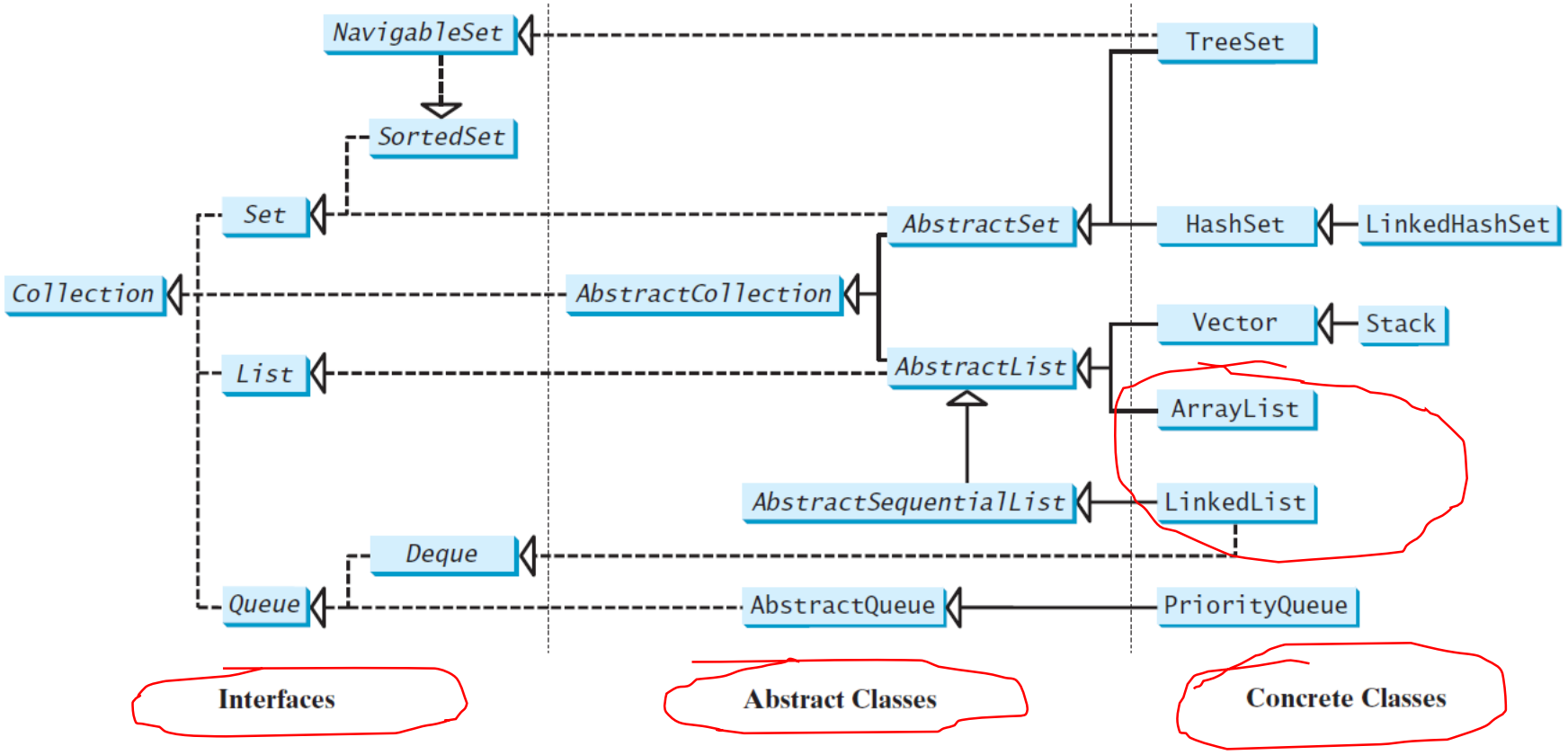


Adds the specified object to the list.
Returns true if this list iterator has more elements when traversing backward.
Returns the index of the next element.
Returns the previous element in this list iterator.
Returns the index of the previous element.
Replaces the last element returned by the previous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

Questions?

- The List interface
  - Methods and type hierarchy
- How much do you remember?

Lists: ArrayList and LinkedList



ArrayList and LinkedList

- List
  - stores elements in a sequential order,
  - allows the user to specify where the element is stored.
  - the user may be able to access the elements by index.
- ArrayList
  - Efficient random access: access it like an array: access any element at (almost) equal amount of time and efficiently (called near-constant-time positional access)
  - In general, costly to remove and insert elements
- LinkedList
  - Efficient to add and remove elements anywhere
  - Costly random access (does not provide near-constant-time positional access)

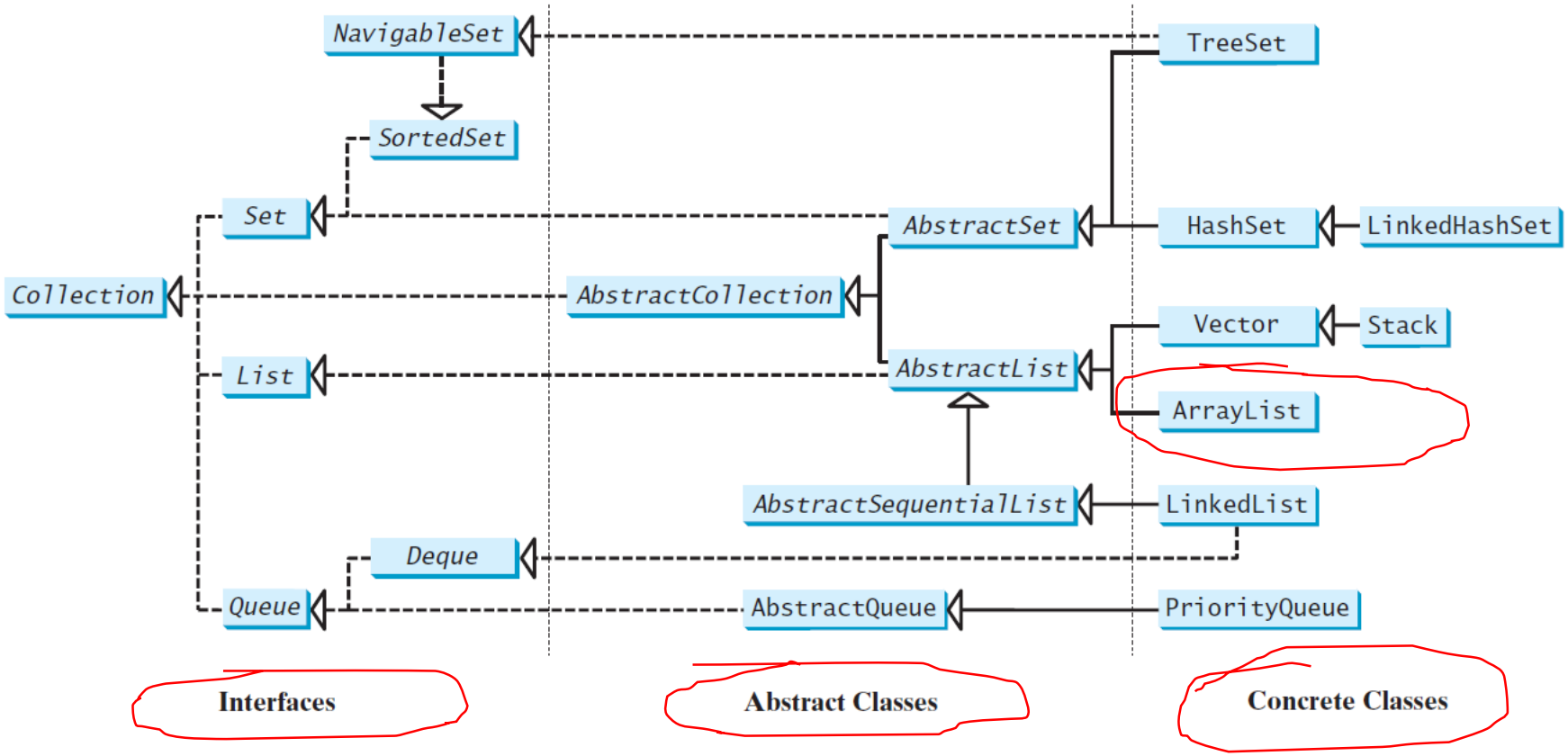
Array, ArrayList, and LinkedList: When to Use?

- ArrayList: if your application needs to support random access through an index without inserting or removing elements from any place other than the end
- LinkedList: if your application requires the insertion or deletion of elements from any place in the list
- Array: an array is fixed in size while a list can grow or shrink dynamically. If your application knows the size and does not require insertion or deletion of elements, the most efficient data structure is the array.

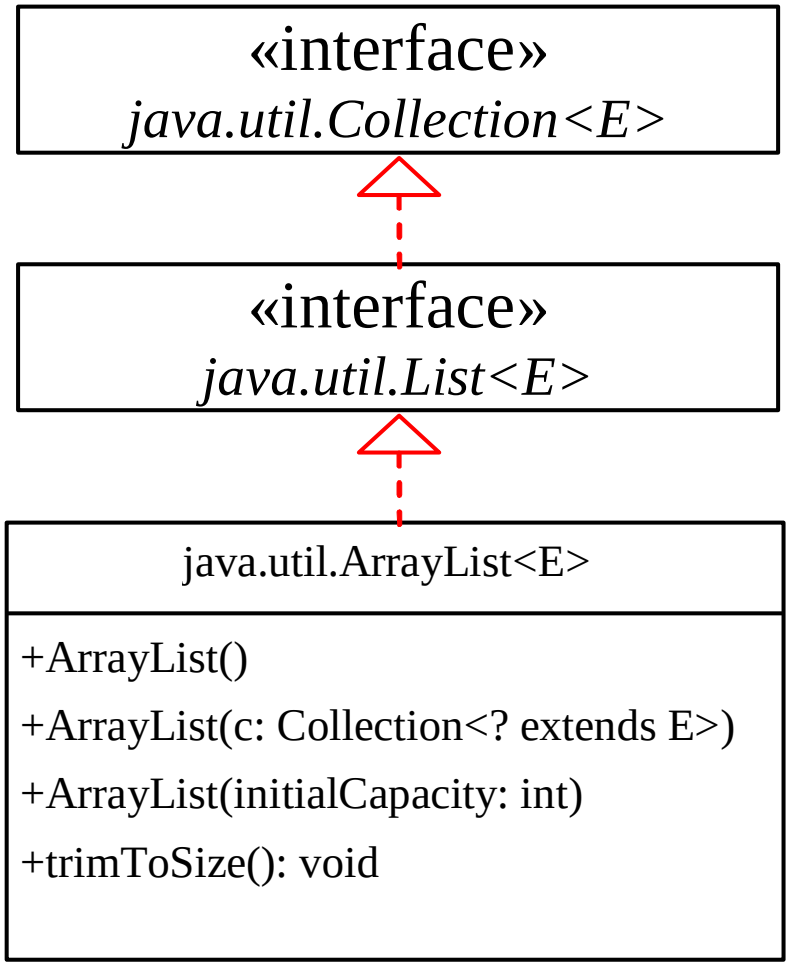
Questions?

- Concept and type hierarchy of ArrayList and Linked List
- When to use array, ArrayList, and LinkedList?

The ArrayList



ArrayList: java.util.ArrayList



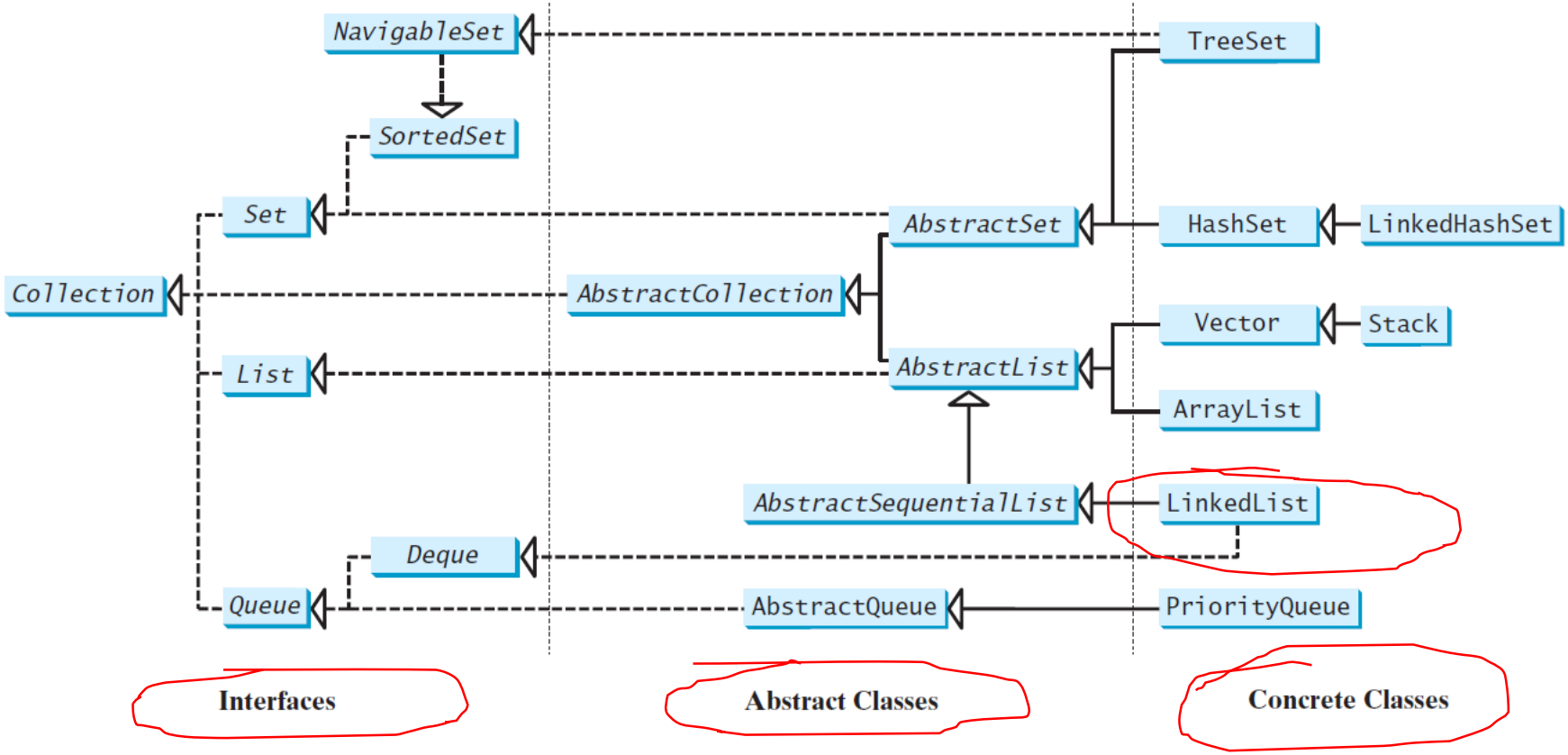
Creates an empty list with the default initial capacity.

Creates an array list from an existing collection.

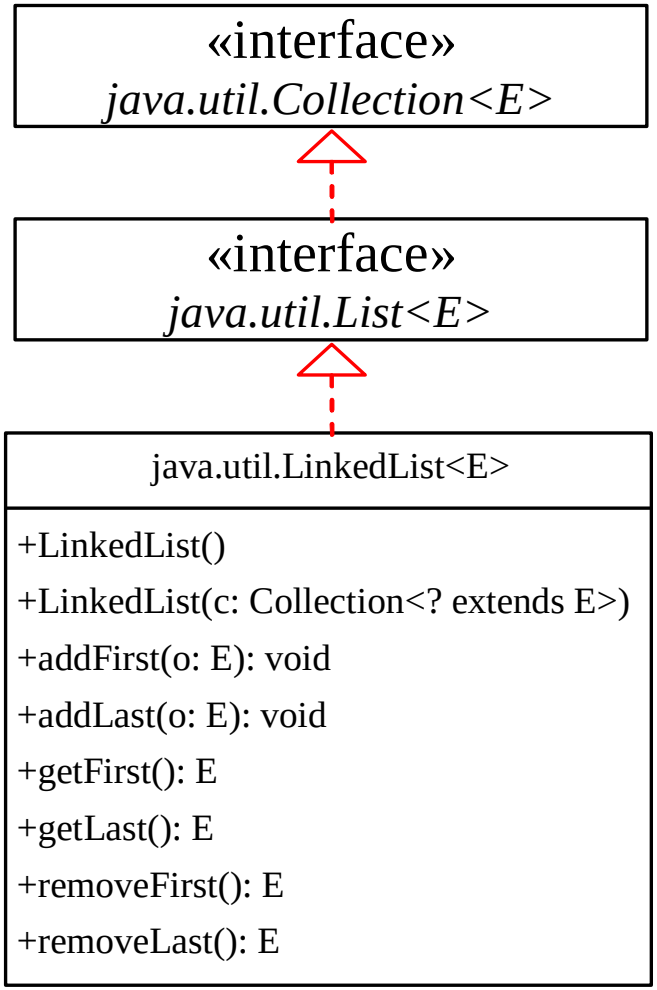
Creates an empty list with the specified initial capacity.

Trims the capacity of this ArrayList instance to be the list's current size.

The LinkedList



LinkedList: java.util.LinkedList



Creates a default empty linked list.

Creates a linked list from an existing collection.

Adds the object to the head of this list.

Adds the object to the tail of this list.

Returns the first element from this list.

Returns the last element from this list.

Returns and removes the first element from this list.

Returns and removes the last element from this list.

Explore ArrayList and LinkedList: Examples

- Random access cost
- Insertion and deletion cost
- Iterating lists
- Updating objects in lists

Questions?

- Use ArrayList and LinkedList
  - Access: random and sequential
  - Use loops and iterators
  - Update objects in lists