Marwadi
education foundation

Unit – 10

List, Stacks, Queues and Priority Queues

Prepared By

Prof. Ravikumar Natarajan

Assistant Professor, CE Dept.

KNOWLEDGE IS THE CURRENCY
FOR THE 21st CENTURY

Contents

- Collection
- Iterators
- Lists
- The Comparator interface
- Static methods for list and collections
- Vector and Stack classes
- Queues and priority Queues

Introduction

- A data structure is a collection of data organized in some fashion.
- The structure not only stores data but also supports operations for accessing and manipulating the data.
- In object-oriented thinking, a data structure, also known as a container or container object, is an object that stores other objects, referred to as data or elements.
- To define a data structure is essentially to define a class.
- The class for a data structure should use data fields to store data and provide methods to support such operations as search, insertion, and deletion.

Introduction

- To create a data structure is therefore to create an instance from the class.
- You can then apply the methods on the instance to manipulate the data structure, such as inserting an element into or deleting an element from the data structure.
- **Choosing the best data structures and algorithms for a particular task is one of the keys to developing high-performance software.**

Collections

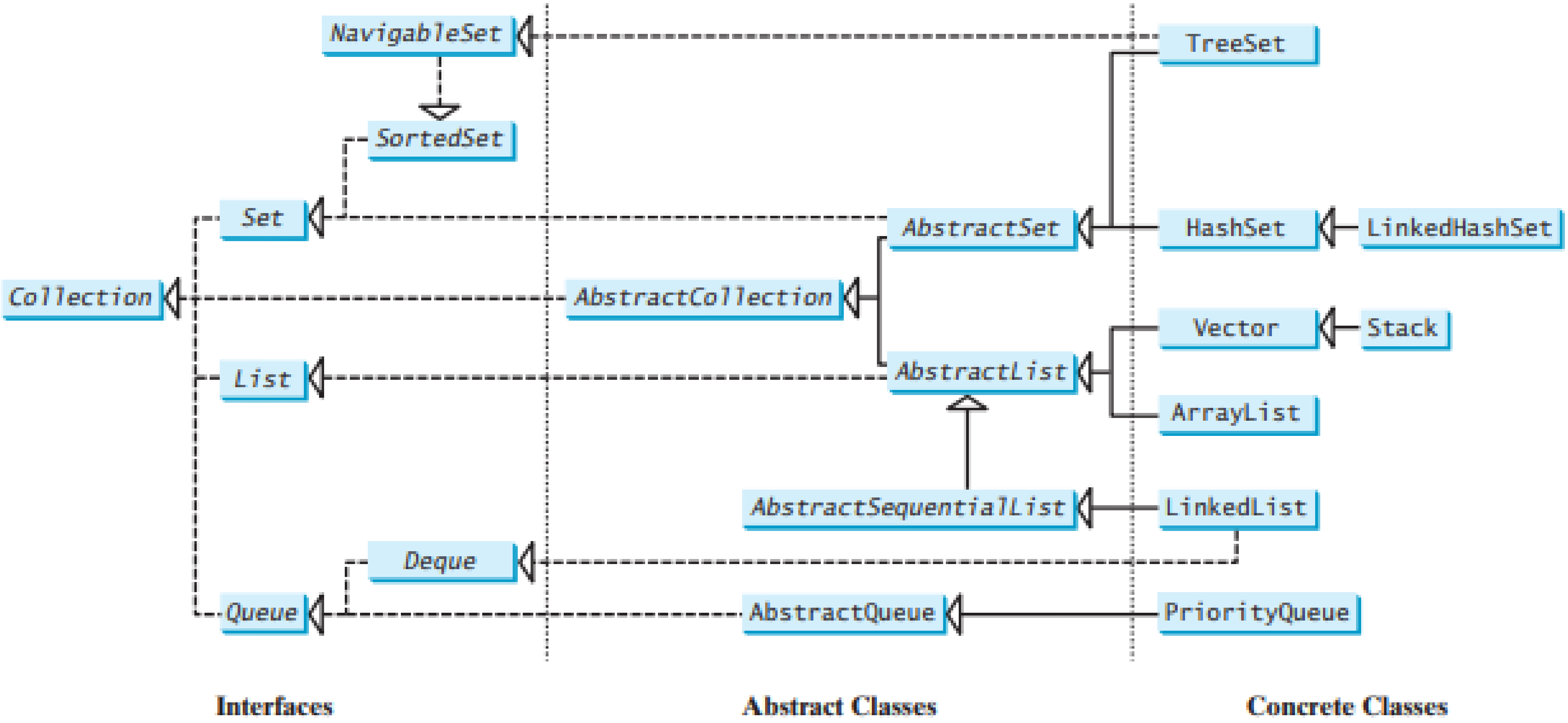
- The Java Collections Framework supports two types of containers:
  - One for storing a collection of elements is simply called a collection.
  - The other, for storing key/value pairs, is called a map (efficient for searching).
- The common features of collections are defined in the interfaces, and implementations are provided in concrete classes (normal class).
- **The Collection interface defines the common operations for lists, vectors, stacks, queues, priority queues, and sets.**

Collections

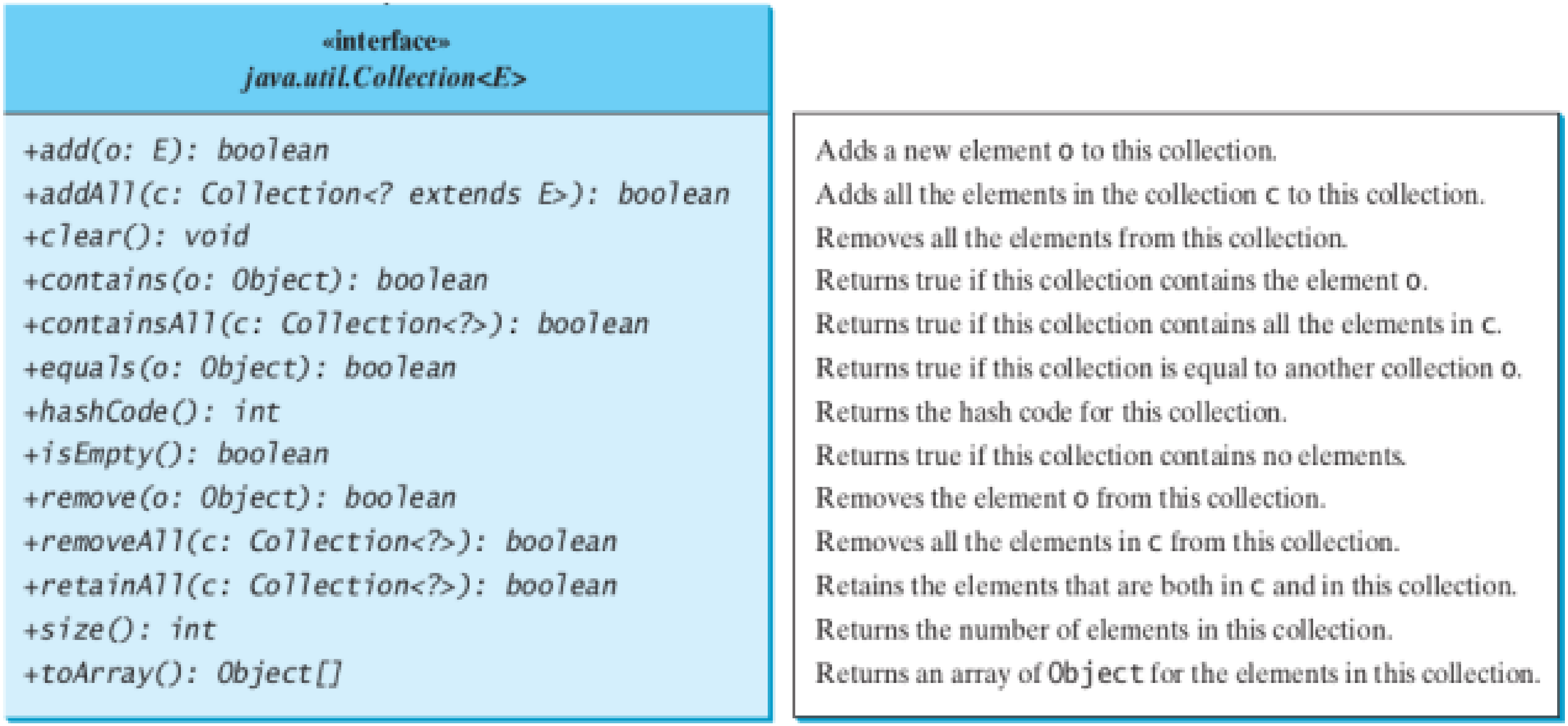
- **Sets** store a group of nonduplicate elements.
- **Lists** store an ordered collection of elements.
- **Stacks** store objects that are processed in a last-in, first-out **LIFO** fashion.
- **Queues** store objects that are processed in a first-in, first-out **FIFO** fashion.
- **PriorityQueues** store objects that are processed in the order of their priorities.

Collections

- A collection is a container that stores objects.



Collection Interface



«interface»
java.util.Collection<E>

+add(o: E): boolean	Adds a new element o to this collection.
+addAll(c: Collection<? extends E>): boolean	Adds all the elements in the collection c to this collection.
+clear(): void	Removes all the elements from this collection.
+contains(o: Object): boolean	Returns true if this collection contains the element o.
+containsAll(c: Collection<?>): boolean	Returns true if this collection contains all the elements in c.
+equals(o: Object): boolean	Returns true if this collection is equal to another collection o.
+hashCode(): int	Returns the hash code for this collection.
+isEmpty(): boolean	Returns true if this collection contains no elements.
+remove(o: Object): boolean	Removes the element o from this collection.
+removeAll(c: Collection<?>): boolean	Removes all the elements in c from this collection.
+retainAll(c: Collection<?>): boolean	Retains the elements that are both in c and in this collection.
+size(): int	Returns the number of elements in this collection.
+toArray(): Object[]	Returns an array of Object for the elements in this collection.

Iterators

- Iterator is a classic design pattern for walking through a data structure without having to expose the details of how data is stored in the data structure.

- The Collection interface extends the Iterable interface.

- The Iterable interface defines the iterator method, which returns an iterator.
- The Collection interface contains the methods for manipulating the elements in a collection, and you can obtain an iterator object for traversing elements in the collection.

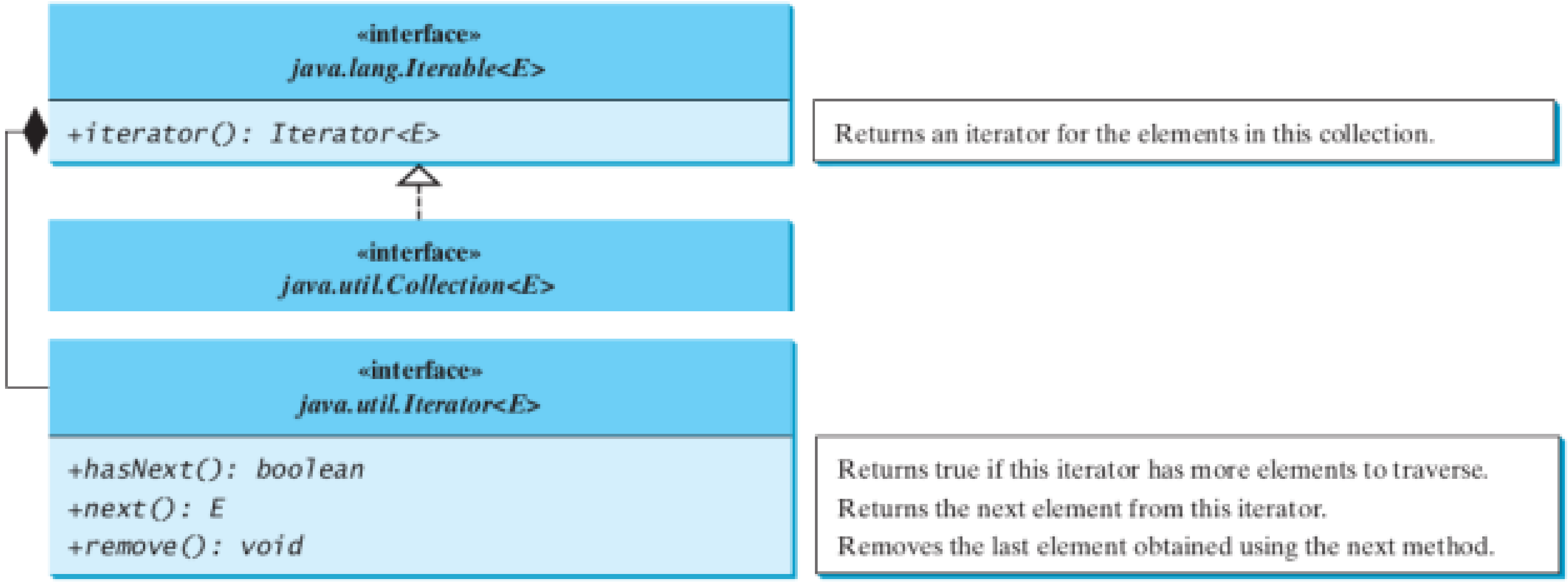
Iterators

- Iterator is a classic design pattern for walking through a data structure without having to expose the details of how data is stored in the data structure.

- The Collection interface extends the Iterable interface.

- The Iterable interface defines the iterator method, which returns an iterator.
- The Collection interface contains the methods for manipulating the elements in a collection, and you can obtain an iterator object for traversing elements in the collection.

Iterators



«interface»
java.lang.Iterable<E>

`+iterator(): Iterator<E>`

Returns an iterator for the elements in this collection.

«interface»
java.util.Collection<E>

«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.

Iterators

```
import java.util.*;

 public class Main {
 public static void main(String[] args) {
 Collection<String> collection = new ArrayList<>();
 collection.add("New York");
 collection.add("Atlanta");
 collection.add("Dallas");
 collection.add("Madison");

Iterator<String> iterator = collection.iterator();
while (iterator.hasNext()) {
System.out.print(iterator.next().toUpperCase() + " ");
}
System.out.println();  }  }
```

Output:

```
NEW YORK ATLANTA DALLAS MADISON
```

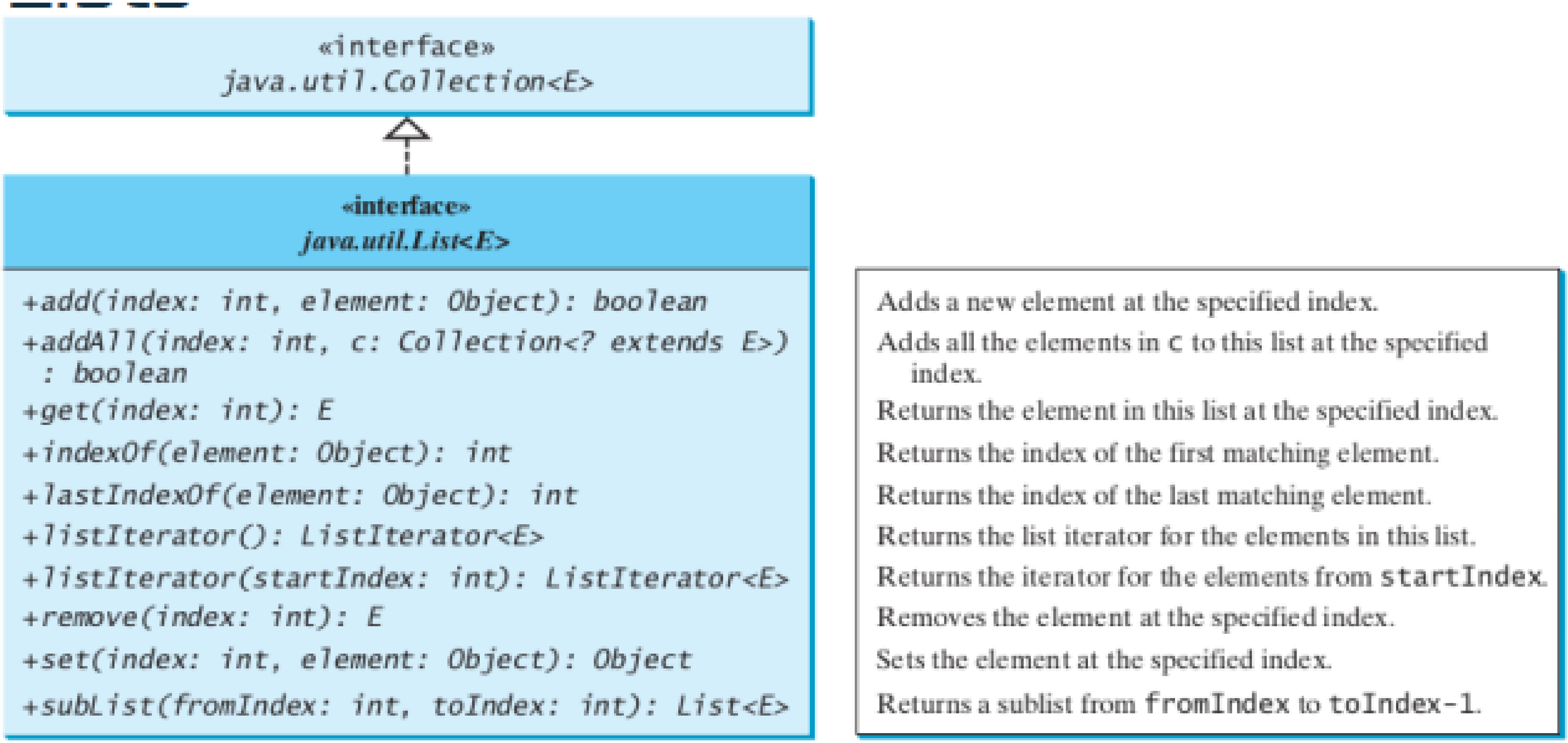
Lists

- The List interface extends the Collection interface and defines a collection for storing elements in a sequential order.

- **To create a list, use one of its two concrete classes: ArrayList or LinkedList.**

- The List interface adds position-oriented operations, as well as a new list iterator that enables the user to traverse the list bidirectionally.

Lists



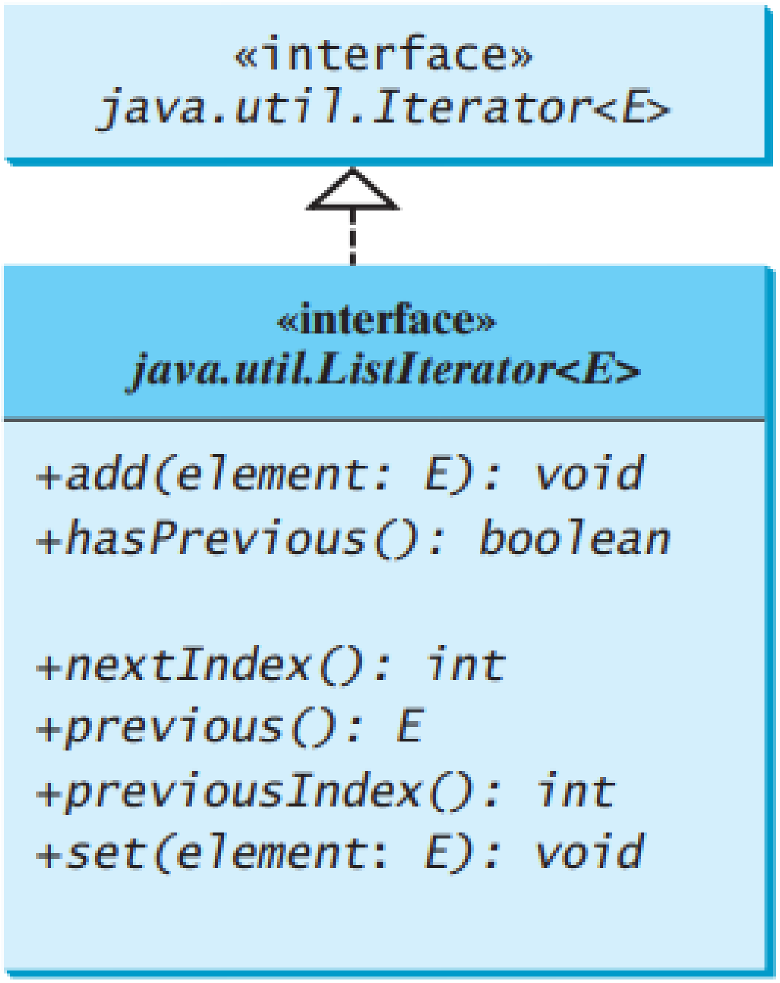
«interface»
java.util.Collection<E>

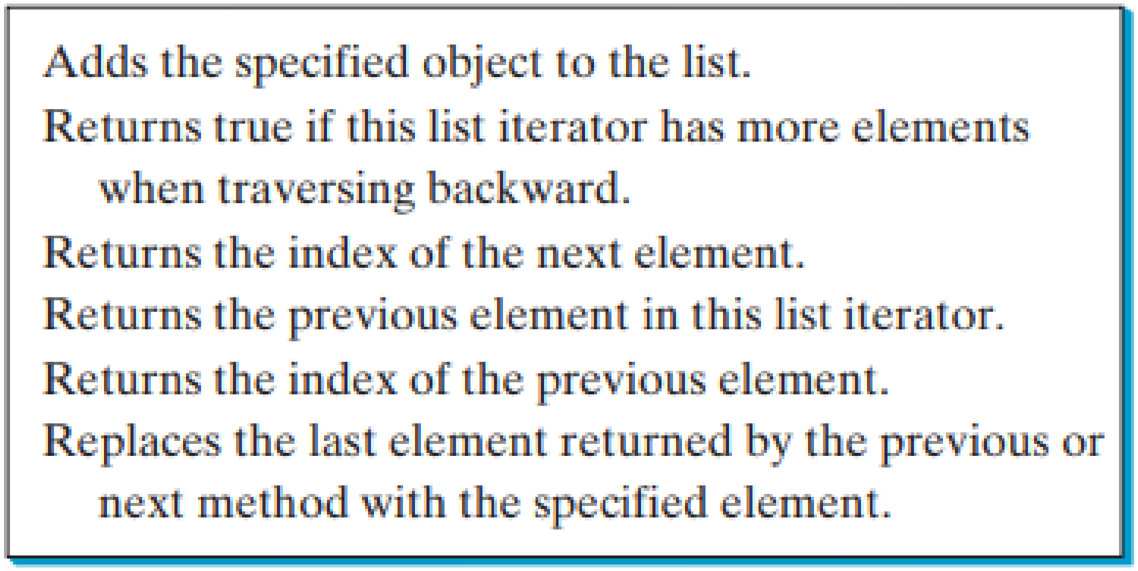
«interface»
java.util.List<E>

+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.

List Iterator

- The listIterator() or listIterator(startIndex) method returns an instance of ListIterator.
- The ListIterator interface extends the Iterator interface to add bidirectional traversal of the list.





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.

Iterator vs List Iterator

- Iterator can traverse only in forward direction whereas ListIterator traverses both in forward and backward directions.

- Using Iterator we cannot modify or replace elements present in Collection. But, ListIterator can modify or replace elements with the help of set(E e).

- **Iterator methods are next(), remove(), hasNext()** whereas ListIterator methods are next(), previous(), hasNext(), hasPrevious(), add(E e).

The ArrayList and LinkedList Classes

- The ArrayList class and the LinkedList class are two concrete implementations of the List interface.
- ArrayList stores elements in an array. The array is dynamically created.
- If the capacity of the array is exceeded, a larger new array is created and all the elements from the current array are copied to the new array.
- LinkedList stores elements in a linked list. A list can grow or shrink dynamically.
- Use of these depends on your specific needs.
- **List or ArrayList when to choose?**
- If you need to support random access through an index without inserting or removing elements at the beginning of the list, **ArrayList** offers the most efficient collection.

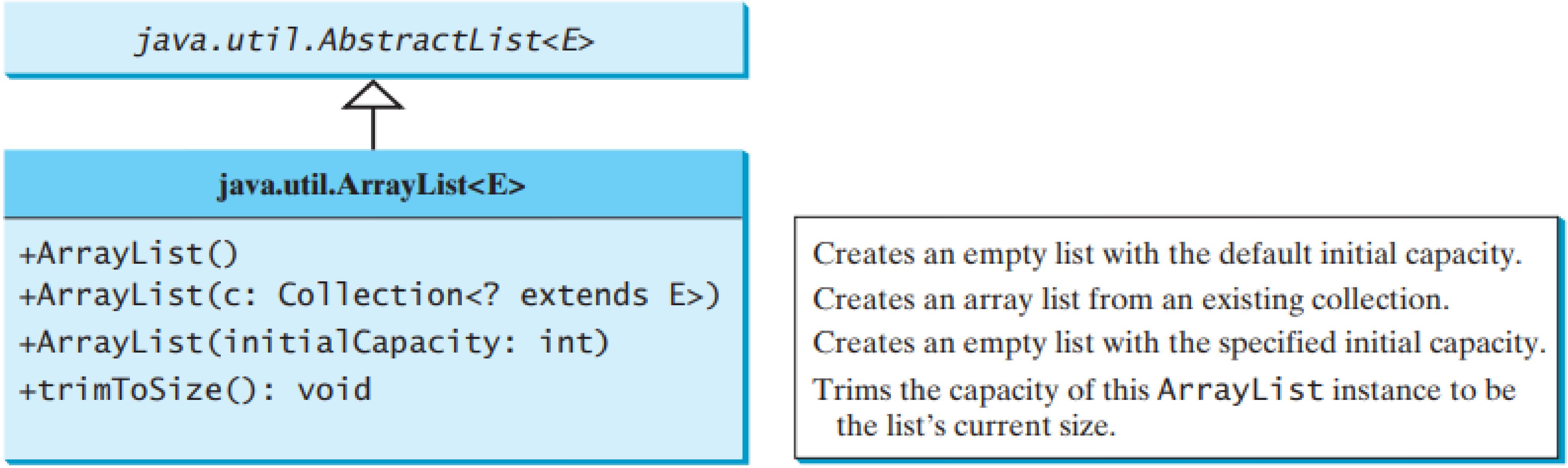
- **If, your application requires the insertion or deletion of elements at the beginning of the list, you should choose LinkedList.**

The ArrayList and LinkedList Classes

- ArrayList is a resizable-array implementation of the List interface.
- It also provides methods for manipulating the size of the array used internally to store the list.
- Each ArrayList instance has a capacity, which is the size of the array used to store the elements in the list.

- As elements are added to an ArrayList, its capacity grows automatically.
- An ArrayList does not automatically shrink. You can use the **trimToSize()** method to reduce the array capacity to the size of the list.

The ArrayList Class



java.util.AbstractList<E>

java.util.ArrayList<E>

```
+ArrayList()
+ArrayList(c: Collection<? extends E>)
+ArrayList(initialCapacity: int)
+trimToSize(): void
```

Creates an empty list with the default initial capacity.

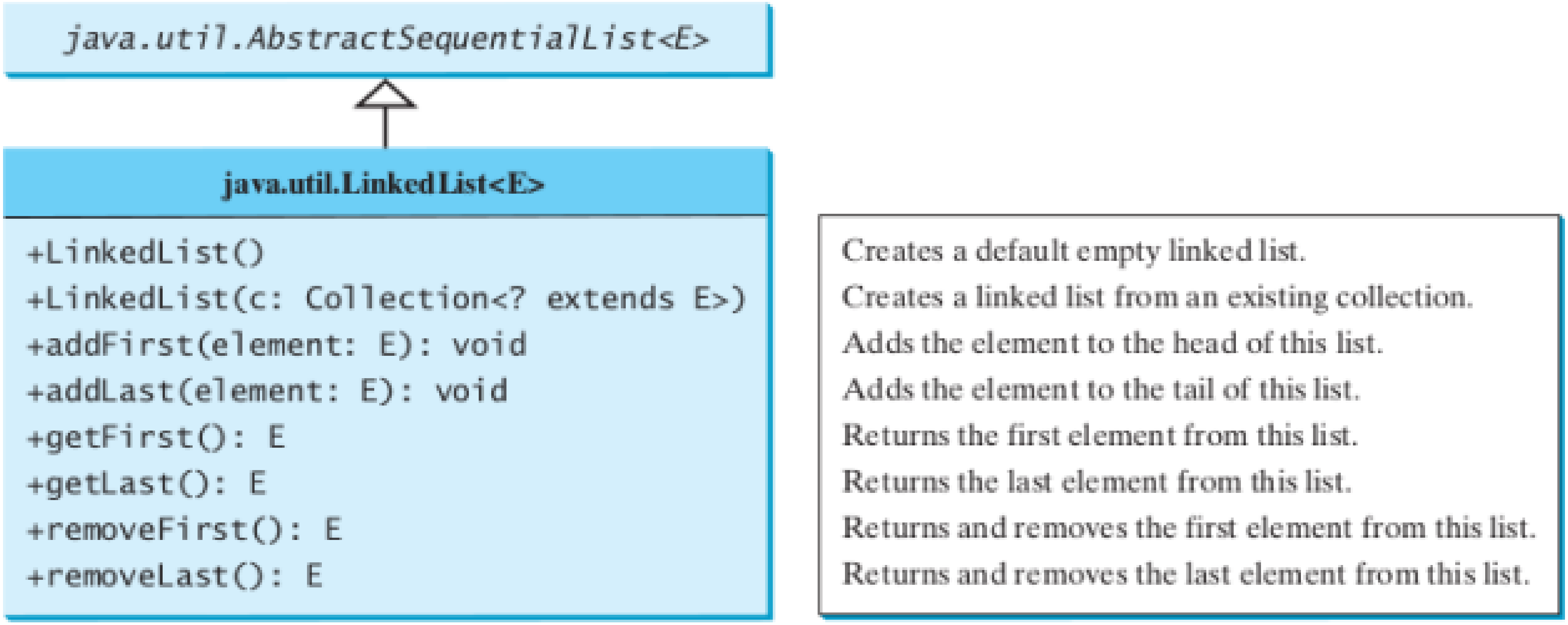
Creates an array list from an existing collection.

Creates an empty list with the specified initial capacity.

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

Refer to the example in Slide number: 12

The LinkedList Class



java.util.AbstractSequentialList<E>

java.util.LinkedList<E>

```
+LinkedList()
+LinkedList(c: Collection<? extends E>)
+addFirst(element: E): void
+addLast(element: E): void
+getFirst(): E
+getLast(): E
+removeFirst(): E
+removeLast(): E
```

Creates a default empty linked list.
Creates a linked list from an existing collection.
Adds the element to the head of this list.
Adds the element 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.

The LinkedList Class

```
import java.util.*;
class Main{
  public static void main(String[] args) {
LinkedList<String> list = new LinkedList<String>();
 list.add("Delhi"); list.add(" Moscow");
 list.add("NewYork");
System.out.println(list);

list.remove("Moscow");
 System.out.println("After Deleting Elements: \n"+ list);

list.remove(1);
System.out.println("After Deleting Elements: \n"+ list);

list.addFirst(“First”);
List.removeLast();
System.out.println("After Deleting Elements: \n"+ list);
 } }
```

Output:

```
[Delhi,  Moscow, NewYork]
After Deleting Elements:
[Delhi,  Moscow, NewYork]
After Deleting Elements:
[Delhi, NewYork]
After Add first and remove last Elements:
[First, Delhi]
```

The ArrayList and LinkedList Classes

Extra Example for reference

```
import java.util.*;

 public class Main {
 public static void main(String[] args) {
 List<Integer> arrayList = new ArrayList<>();
 arrayList.add(1); // 1 is autoboxed to new Integer(1)
 arrayList.add(2);
 arrayList.add(3);
 arrayList.add(1);
arrayList.add(4);
arrayList.add(0, 10);
arrayList.add(3, 30);

System.out.println("A list of integers in the array list:");
System.out.println(arrayList);

 LinkedList<Object> linkedList = new
LinkedList<>(arrayList);
 linkedList.add(1, "red");
 linkedList.removeLast();
 linkedList.addFirst("green");

System.out.println("Display the linked list forward:");
ListIterator<Object> listIterator = linkedList.listIterator();
while (listIterator.hasNext()) {
System.out.print(listIterator.next() + " ");
}
System.out.println();

System.out.println("Display the linked list backward:");
listIterator = linkedList.listIterator(linkedList.size());
while (listIterator.hasPrevious()) {
System.out.print(listIterator.previous() + " ");
}
}
}
```

The Comparator interface

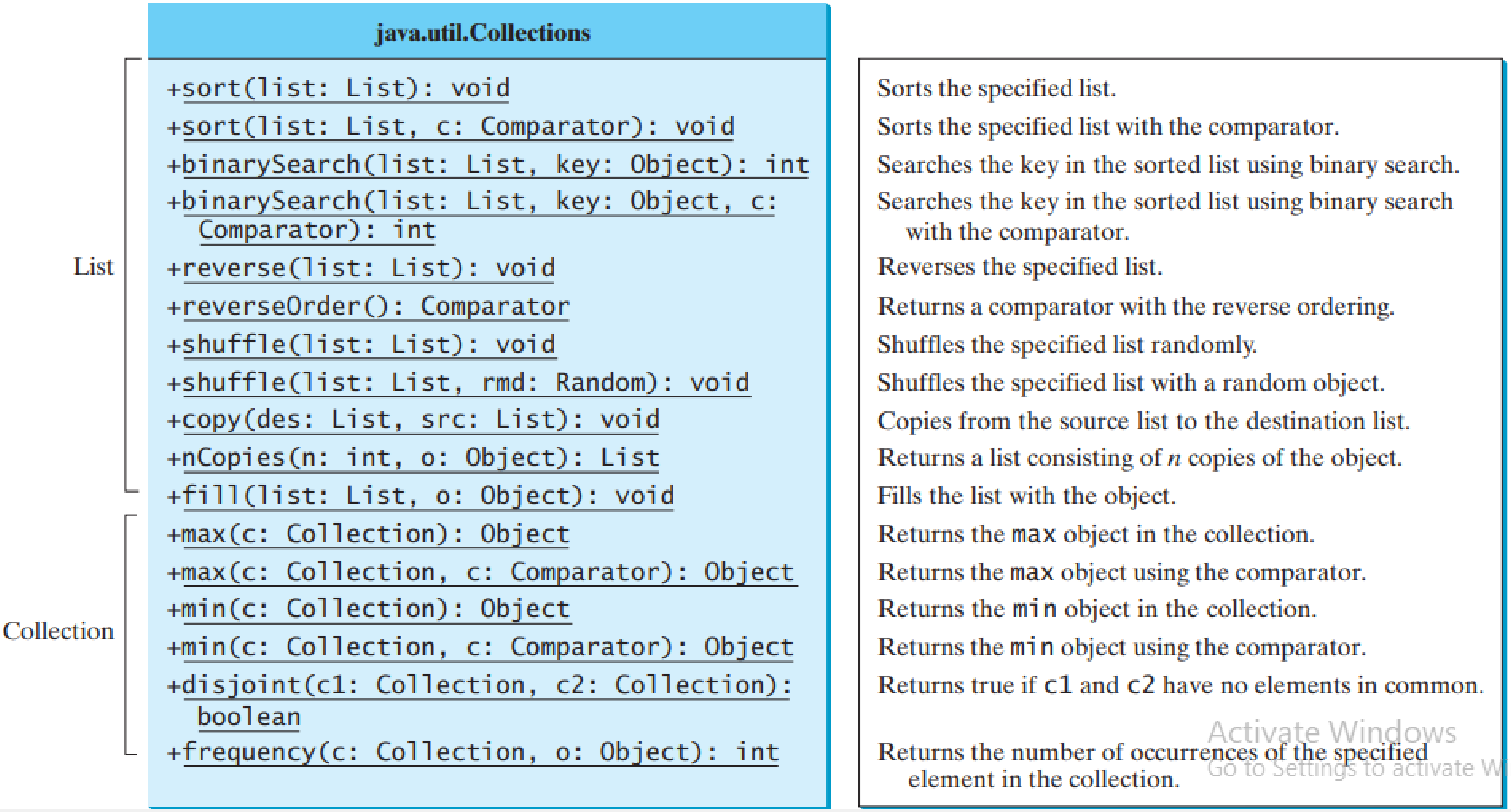
- Already discussed. Refer previous chapter

Static Methods for List and Collections

- The Collections class contains static methods to perform common operations in a collection and a list.

- The Collections class contains the sort, binarySearch, reverse, shuffle, copy, and fill methods for lists, and **max, min, disjoint, and frequency methods for collections.**

Static Methods for List and Collections



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.

Vector and Stack Classes

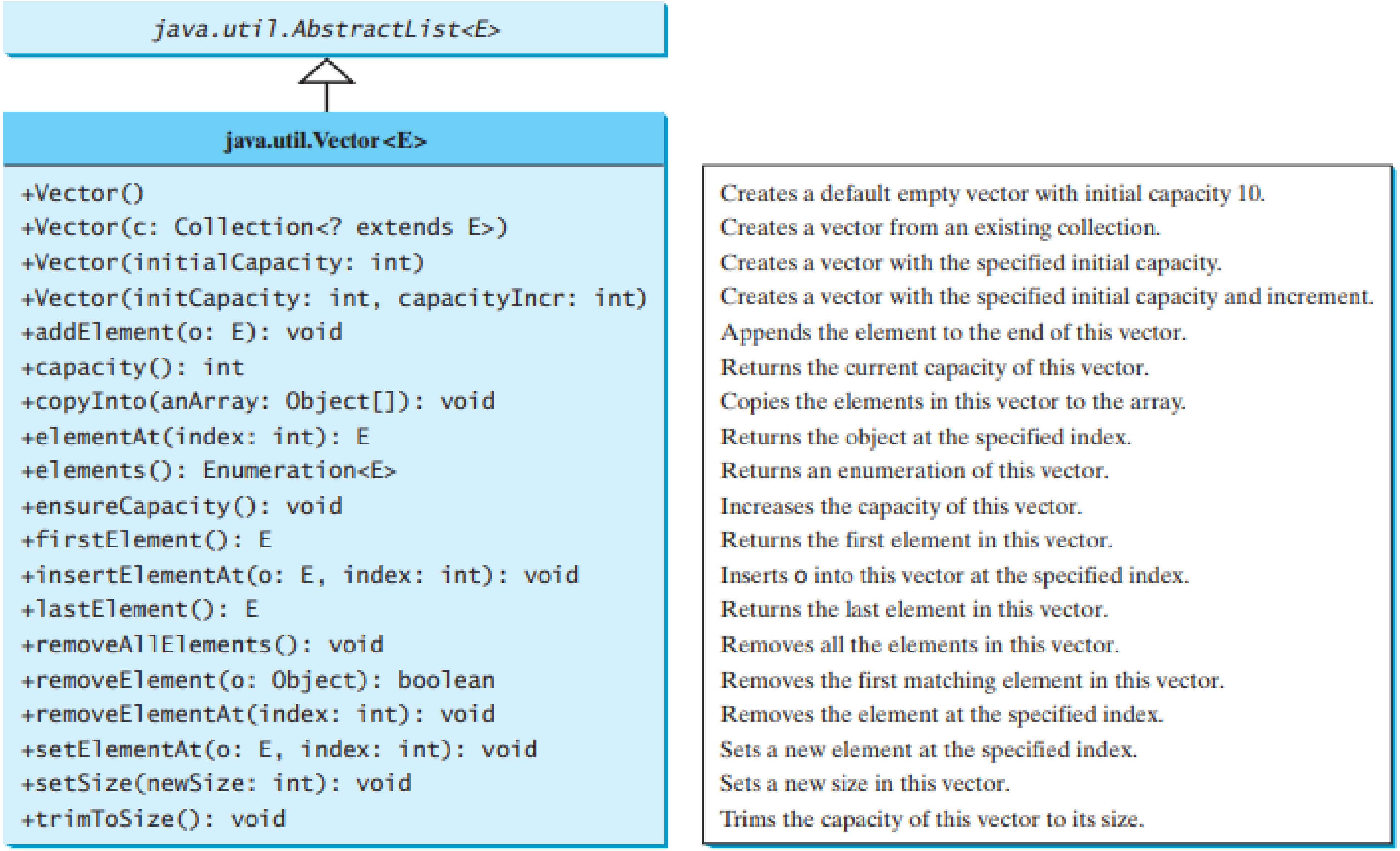
- The Collections class contains static methods to perform common operations in a collection and a list.

- The Collections class contains the sort, binarySearch, reverse, shuffle, copy, and fill methods for lists, and **max, min, disjoint, and frequency methods for collections.**

- **Vector is a subclass of AbstractList, and Stack is a subclass of Vector in the Java API.**

- Vector is the same as ArrayList, except that it contains synchronized methods for accessing and modifying the vector. Synchronized methods can prevent data corruption when a vector is accessed and modified by two or more threads concurrently.

Vector Classes



java.util.AbstractList<E>

java.util.Vector<E>

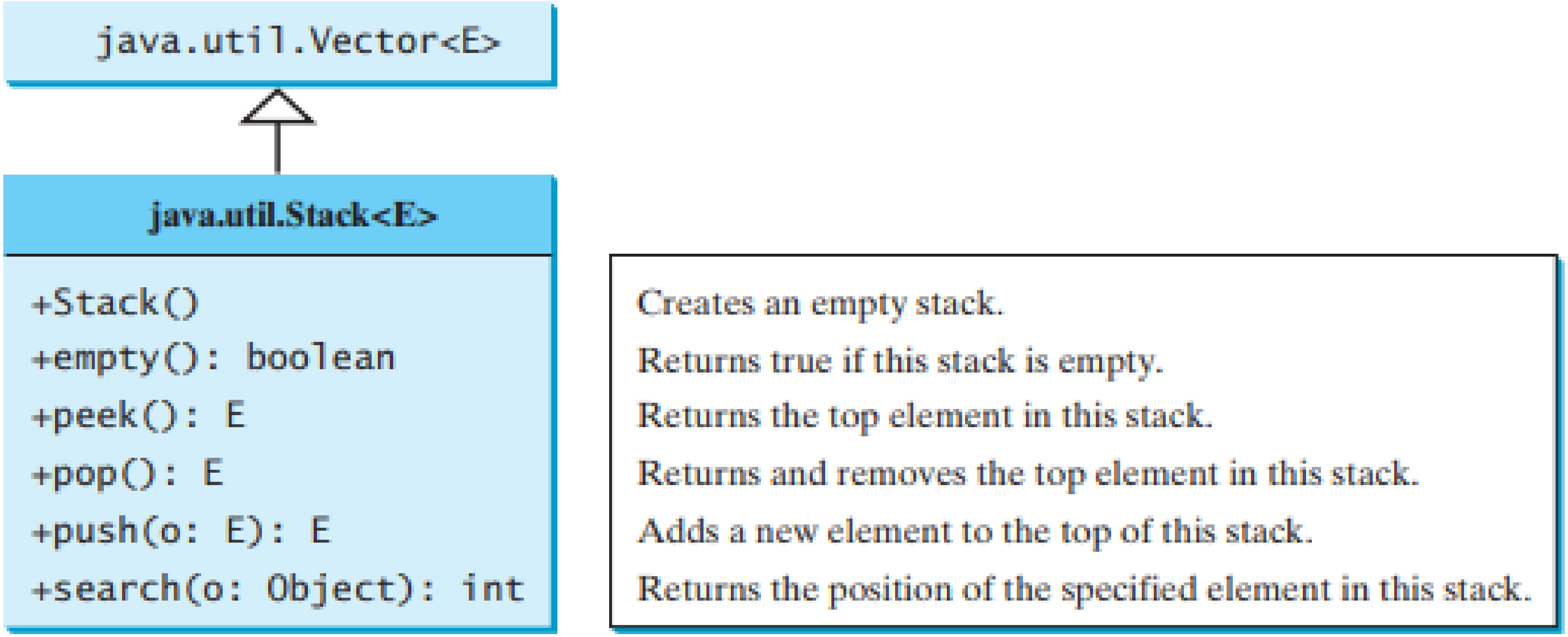
+Vector()	Creates a default empty vector with initial capacity 10.
+Vector(c: Collection<? extends E>)	Creates a vector from an existing collection.
+Vector(initialCapacity: int)	Creates a vector with the specified initial capacity.
+Vector(initCapacity: int, capacityIncr: int)	Creates a vector with the specified initial capacity and increment.
+addElement(o: E): void	Appends the element to the end of this vector.
+capacity(): int	Returns the current capacity of this vector.
+copyInto(anArray: Object[]): void	Copies the elements in this vector to the array.
+elementAt(index: int): E	Returns the object at the specified index.
+elements(): Enumeration<E>	Returns an enumeration of this vector.
+ensureCapacity(): void	Increases the capacity of this vector.
+firstElement(): E	Returns the first element in this vector.
+insertElementAt(o: E, index: int): void	Inserts o into this vector at the specified index.
+lastElement(): E	Returns the last element in this vector.
+removeAllElements(): void	Removes all the elements in this vector.
+removeElement(o: Object): boolean	Removes the first matching element in this vector.
+removeElementAt(index: int): void	Removes the element at the specified index.
+setElementAt(o: E, index: int): void	Sets a new element at the specified index.
+setSize(newSize: int): void	Sets a new size in this vector.
+trimToSize(): void	Trims the capacity of this vector to its size.

Vector Classes

```
// Java Program to Add Elements in Vector Class
import java.util.*;
class Main {
        public static void main(String[] arg)
        {
          Vector v1 = new Vector();
          //Vector<Integer> v2 = new Vector<Integer>();

                v1.add(1);
                v1.add(2);
                v1.add("Hi");
                v1.add("Hello");
                v1.add(3);

                System.out.println("Vector v1 is " + v1);
        }
}
```

Stack Classes

- In the Java Collections Framework, Stack is implemented as an extension of Vector.
- Stack follows LIFO



java.util.Vector<E>

java.util.Stack<E>

+Stack()	Creates an empty stack.
+empty(): boolean	Returns true if this stack is empty.
+peek(): E	Returns the top element in this stack.
+pop(): E	Returns and removes the top element in this stack.
+push(o: E): E	Adds a new element to the top of this stack.
+search(o: Object): int	Returns the position of the specified element in this stack.

Stack Classes

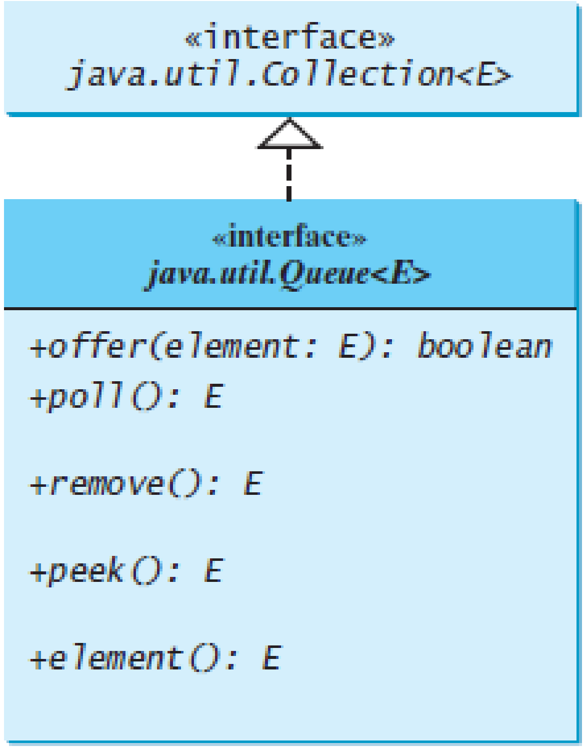
```
// Java program to add & remove the elements in the stack
import java.util.*;

class Main {
        public static void main(String[] args)
        {
                Stack stack1 = new Stack();
                stack1.push(4);
                stack1.push("All");
                stack1.push(“Hello");

                System.out.println(stack1);

                stack1.pop();
                System.out.println(stack1);
        }
}
```

Queue

The **Queue** interface extends **java.util.Collection** with additional insertion, extraction, and inspection operations.

A queue is a first-in, first-out data structure.

Elements are appended to the end of the queue and are removed from the beginning of the queue.



«interface»
java.util.Collection<E>

«interface»
java.util.Queue<E>

```
+offer(element: E): boolean
+poll(): E
+remove(): E
+peek(): E
+element(): E
```

- **offer() and add() both methods can be used to Insert elements into queue.**

Inserts an element into the queue.

Retrieves and removes the head of this queue, or `null` if this queue is empty.

Retrieves and removes the head of this queue and throws an exception if this queue is empty.

Retrieves, but does not remove, the head of this queue, returning `null` if this queue is empty.

Retrieves, but does not remove, the head of this queue, throws an exception if this queue is empty.

Queue

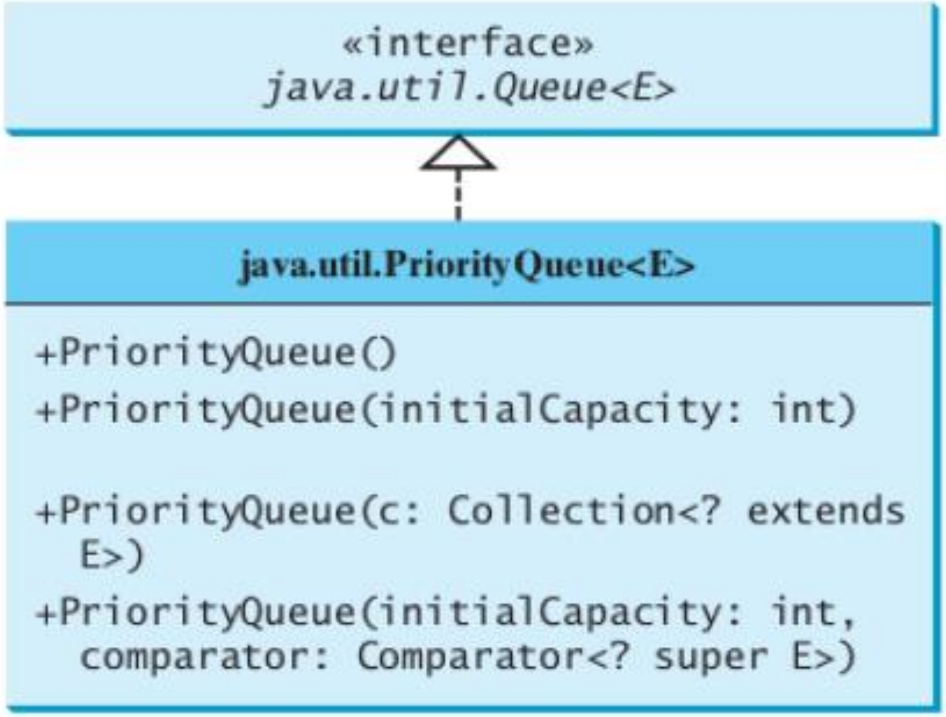
```
import java.util.*;
class Main {
    public static void main(String[] args) {
    Queue<Integer> q = new LinkedList<>();

        // offer elements to the Queue
        q.offer(1); //q.add(1);
        q.offer(2);    q.offer(3);      System.out.println("Queue: " + q);

        // Access elements of the Queue
        int accessedNumber = q.peek();
        System.out.println("Accessed Element: " + accessedNumber);

        // Remove elements from the Queue
        int removedNumber = q.poll();
        System.out.println("Removed Element: " + removedNumber);

        System.out.println("Updated Queue: " + q);     }}
```

Priority Queue

- By default, the priority queue orders its elements according to their natural ordering using Comparable.

- The element with the least value is assigned the highest priority and thus is removed from the queue first.
- If there are several elements with the same highest priority, the tie is broken arbitrarily.
- You can also specify an ordering using Comparator in the constructor PriorityQueue(initialCapacity, comparator).

Priority Queue



«interface»
java.util.Queue<E>

java.util.PriorityQueue<E>

```
+PriorityQueue()
+PriorityQueue(initialCapacity: int)

+PriorityQueue(c: Collection<? extends
  E>)
+PriorityQueue(initialCapacity: int,
  comparator: Comparator<? super E>)
```

Creates a default priority queue with initial capacity 11.

Creates a default priority queue with the specified initial capacity.

Creates a priority queue with the specified collection.

Creates a priority queue with the specified initial capacity and the comparator.

Priority Queue

```
import java.util.*;
class Main{
    public static void main(String args[]){
        Queue<String> pq = new PriorityQueue<>();
        //PriorityQueue<String> pq = new PriorityQueue<>();
        pq.add("abcd");
        pq.add("1234");
        pq.add("java");
        pq.add("catch");
        pq.add("1");
        System.out.println(pq); // Natural Ordering
    }
}
```

Summary

- ArrayList
- Iterator vs ListIterator
- LinkedList
- Vector
- Stack
- Queue
- PriorityQueue

END OF UNIT - 10