Lists, Stacks, Queues, and Priority Queues

Java Collection Framework hierarchy

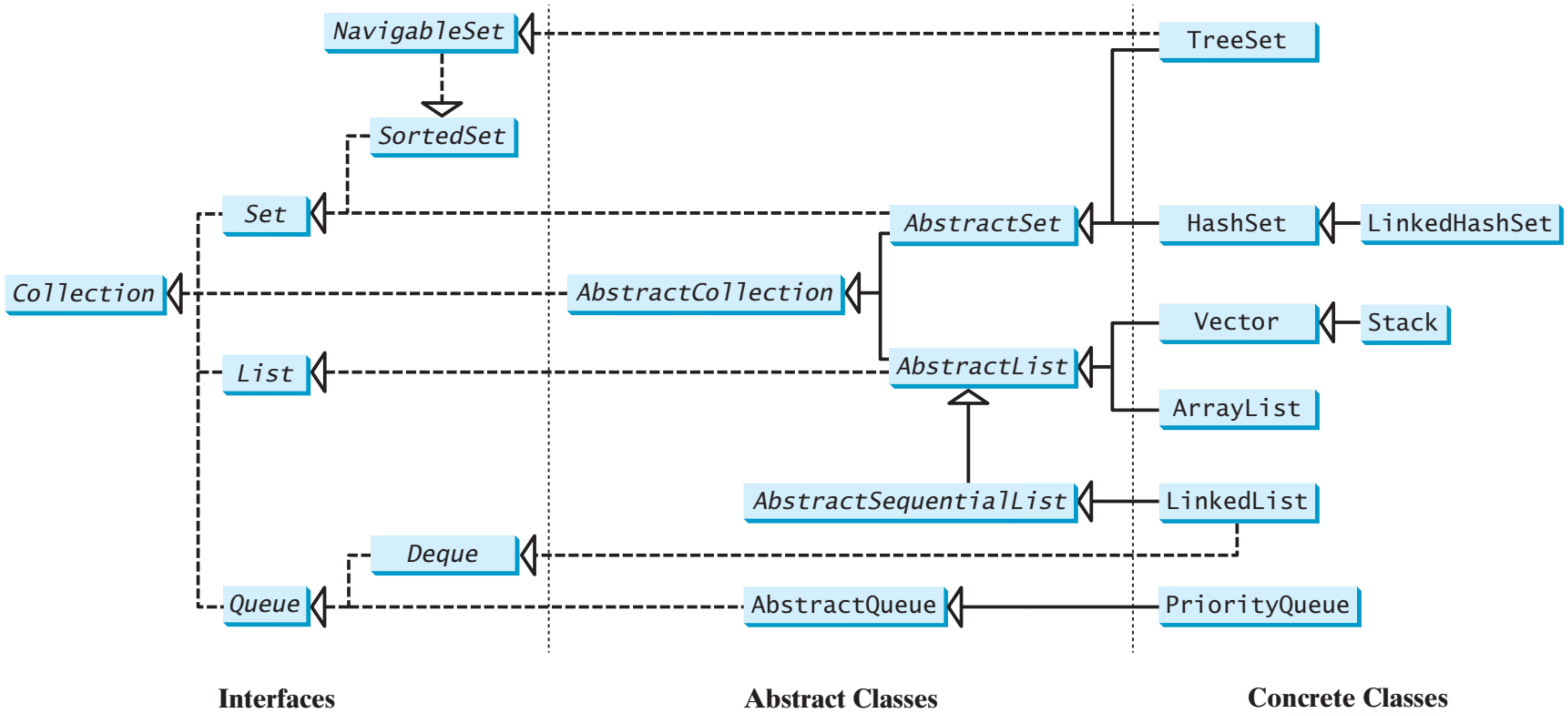
* A *collection* is a container object that holds a group of objects, often referred to as *elements*.

* The Java Collections Framework supports two types of containers:

- One for storing a collection of elements, called a *collection*.
  - 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 fashion.
  - Queues store objects that are processed in a first-in, first-out fashion.
  - PriorityQueues store objects that are processed in the order of their priorities.
- The other, for storing key/value pairs, is called a *map*.
  - Maps are efficient data structures for quickly searching an element using a key.

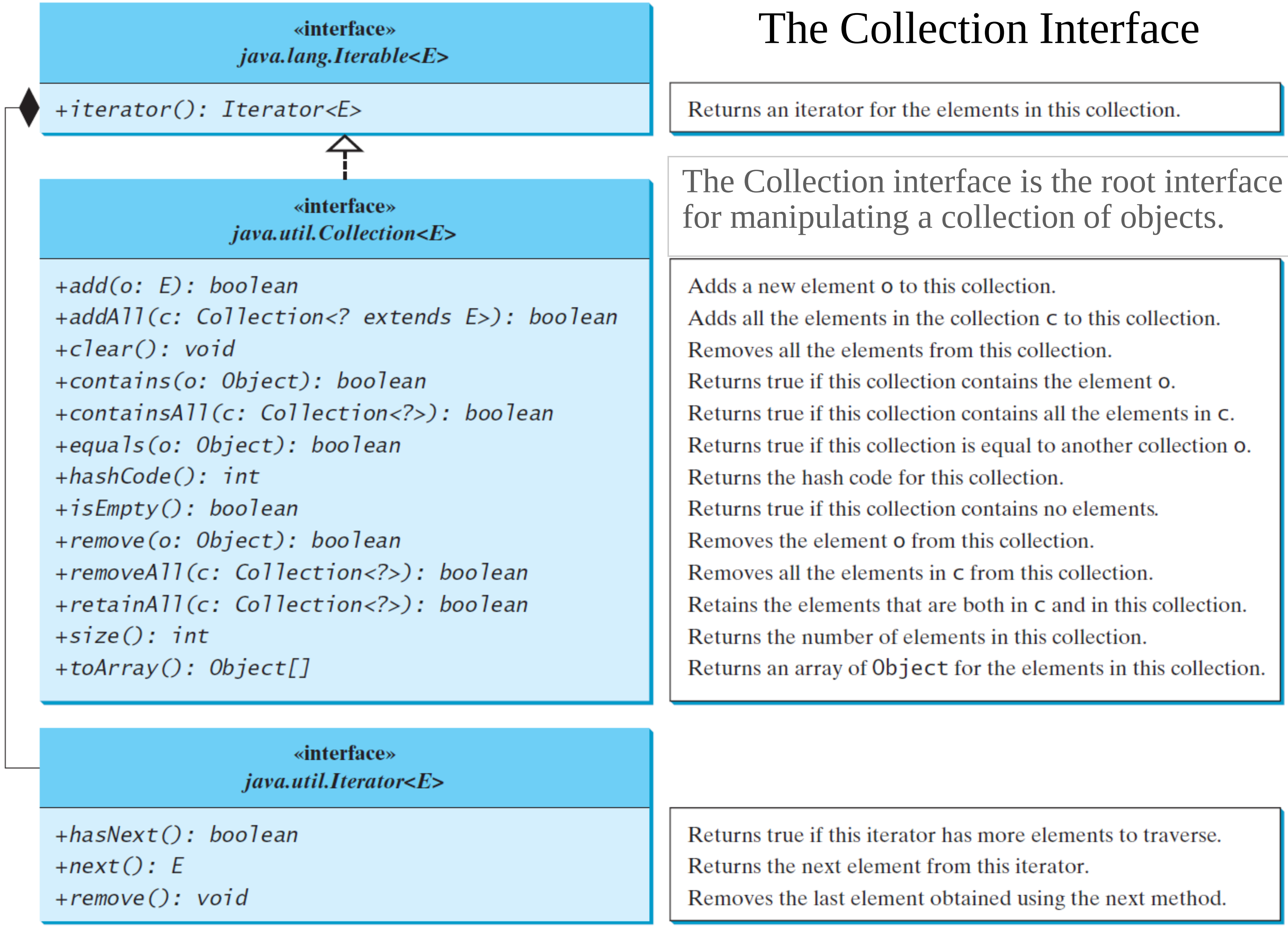
Java Collection Framework hierarchy,cont.

* All the interfaces and classes defined in the Java Collections Framework are grouped in the java.util package.



The Collection Interface

The Collection interface is the root interface for manipulating a collection of objects.



«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`	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.

«interface»
java.util.Iterator<E>

`+hasNext(): boolean`	Returns true if this iterator has more elements to traverse.
`+next(): E`	Returns the next element from this iterator.
`+remove(): void`	Removes the last element obtained using the next method.

The Collection Interface

* Java Collections Framework is an excellent example of using interfaces, abstract classes, and concrete classes.

- The interfaces define the framework.
- The abstract classes provide partial implementation.
- The concrete classes implement the interfaces with concrete data structures.

* The Collection interface is the root interface.

- provides the basic operations for adding and removing elements.
    - add, addAll, remove, removeAll, retainAll, clear()
- The provides various query operations.
    - size, contains, containsAll, isEmpty, toArray()

Iterators

* Iterator is 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 Iterator interface provides a uniform way for traversing elements in various types of collections.
    - provides sequential access to the elements in the collection using the next() method.
    - You can also use the hasNext() method to check whether there are more elements in the iterator,
    - the remove() removes the last element returned by the iterator.

```java
import java.util.*;

public class TestIterator {
  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();
  }
}
```

```
NEW YORK ATLANTA DALLAS MADISON
```

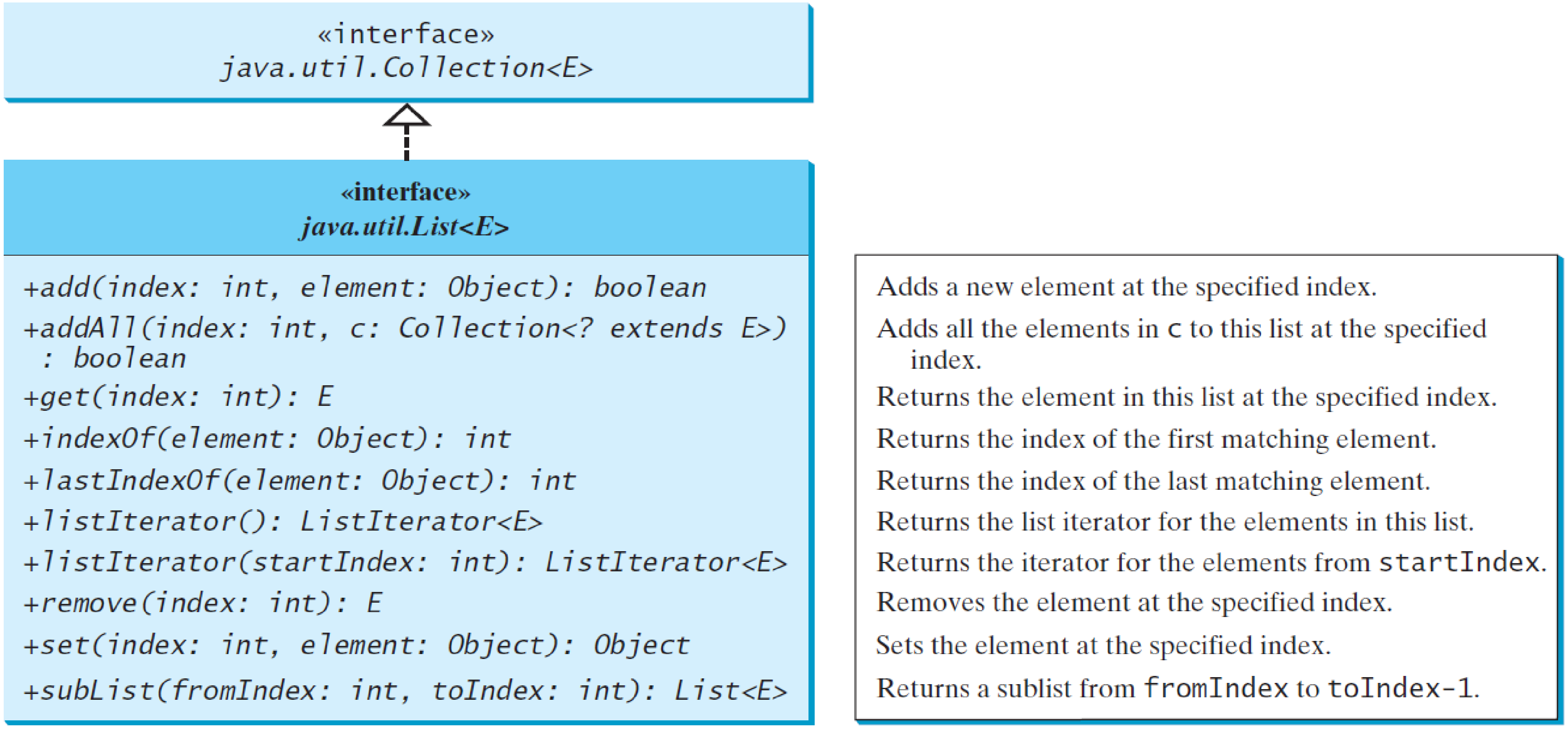
The List Interface

* The List interface extends Collection to define an ordered collection with duplicates allowed.

    - ArrayList and LinkedList are defined under the List interface.

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

The List Interface, cont.



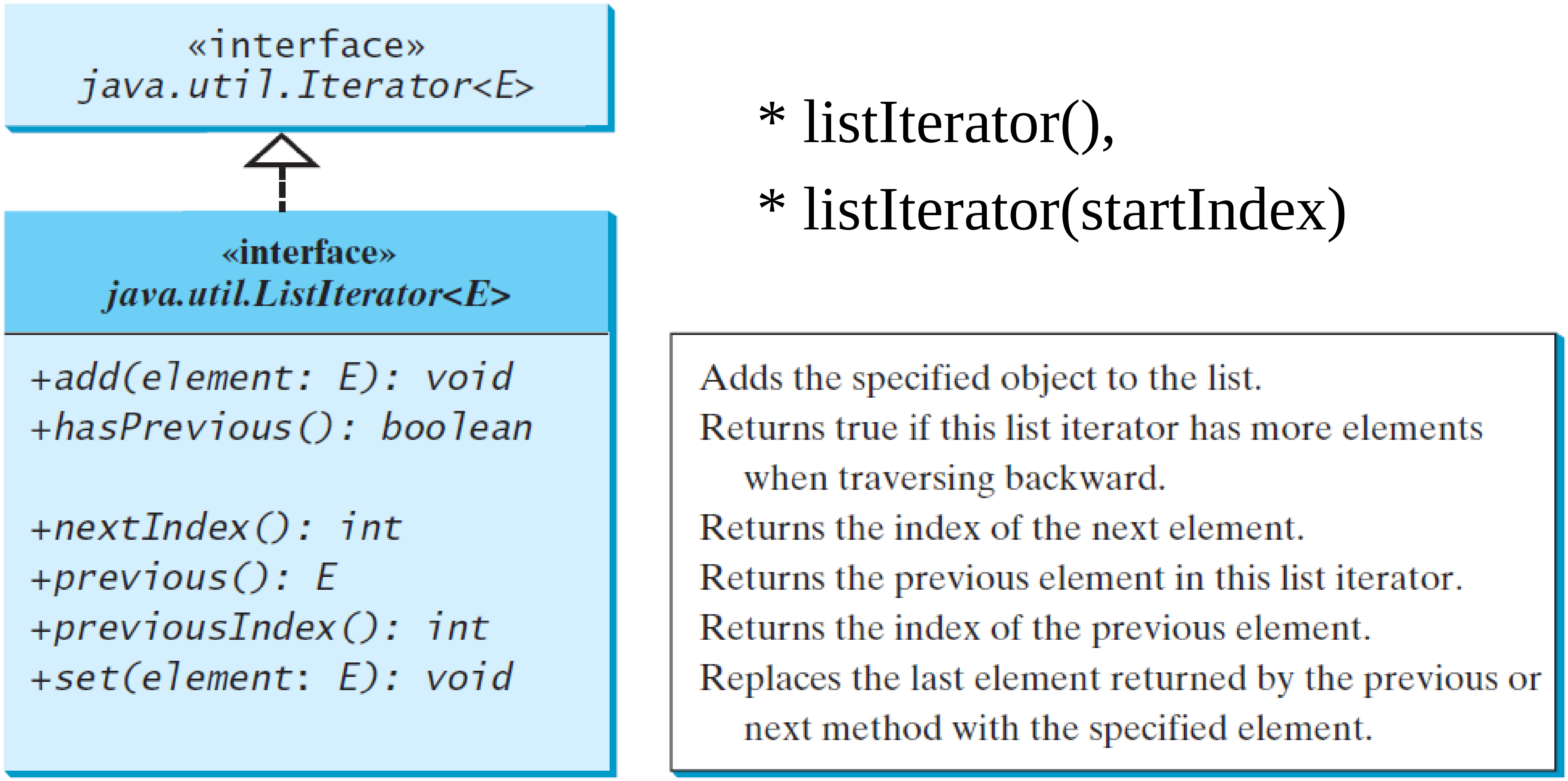
«interface»
java.util.Collection<E>

«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`.

* add(index, element), addAll(index, collection), remove(index), set(index, element), indexOf(element), lastIndexOf(element), subList(fromIndex, toIndex)

The List Iterator



«interface»
java.util.Iterator<E>

«interface»
java.util.ListIterator<E>

+add(element: E): void	Adds the specified object to the list.
+hasPrevious(): boolean	Returns true if this list iterator has more elements when traversing backward.
+nextIndex(): int	Returns the index of the next element.
+previous(): E	Returns the previous element in this list iterator.
+previousIndex(): int	Returns the index of the previous element.
+set(element: E): void	Replaces the last element returned by the previous or next method with the specified element.

* listIterator(),

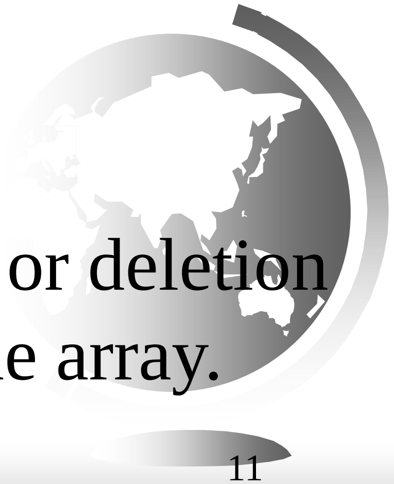
* listIterator(startIndex)

* add(element), next(), previous(), set(element), hasNext(), hasPrevious(), nextIndex(), previousIndex(),

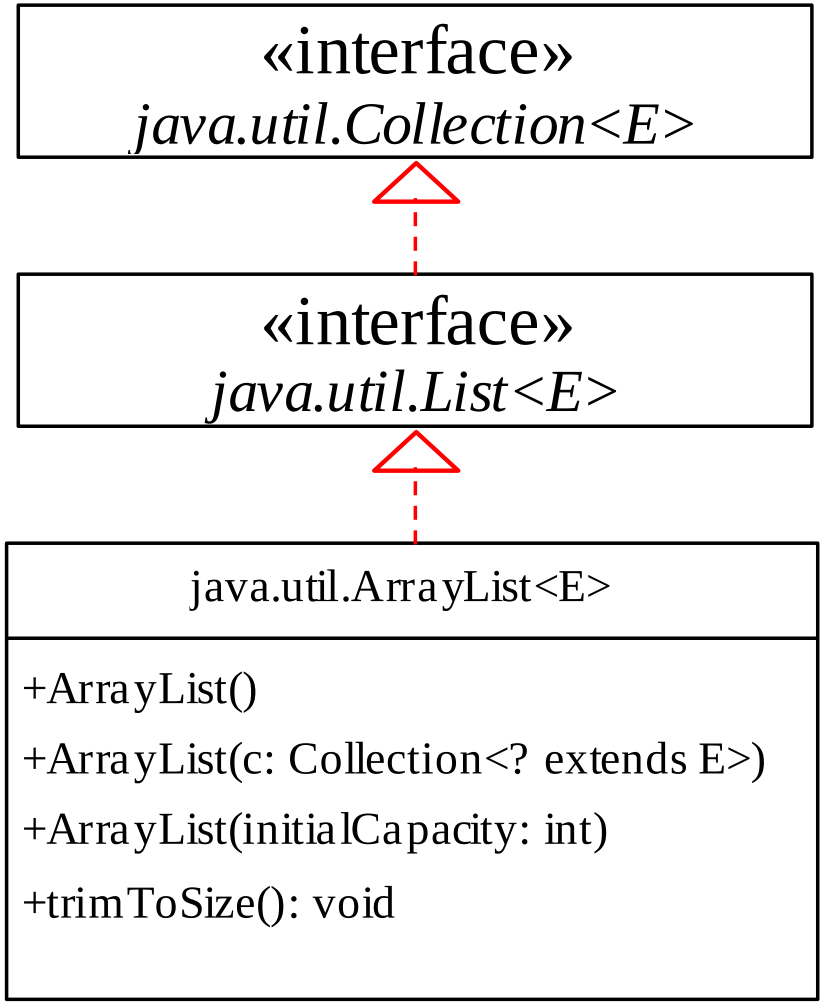
ArrayList and LinkedList

* The ArrayList class and the LinkedList class are concrete implementations of the List interface.

- Which to 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


«interface»
java.util.Collection<E>
«interface»
java.util.List<E>
java.util.ArrayList<E>
+ArrayList()
+ArrayList(c: Collection<? extends E>)
+ArrayList(initialCapacity: int)
+trimToSize(): void

* 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.

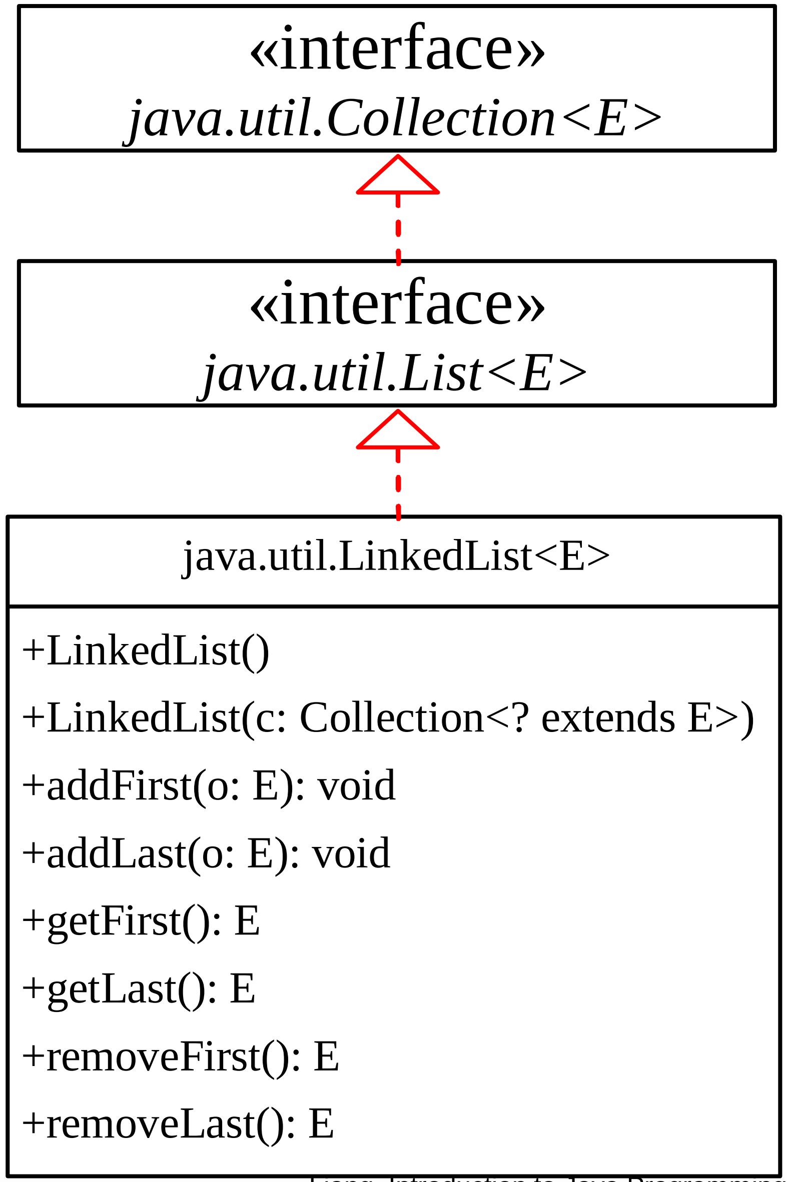
Creates an empty list with the default initial capacity.

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 is a linked list implementation of the List interface.
* In addition to implementing the List interface, this class provides the methods for retrieving, inserting, and removing elements from both ends of the list.

+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.

The Comparator Interface

* You have learned how to compare elements using the Comparable interface.

- Several classes in the Java API, such as String, Date, Calendar, BigInteger, BigDecimal, and all the numeric wrapper classes for the primitive types, implement the Comparable interface.

- The Comparable interface defines the compareTo method, which is used to compare two elements of the same class.

* What if the elements' classes do not implement the Comparable interface? Can these elements be compared?

The Comparator Interface

* you can define a class that implements java.util.Comparator<T> interface and overrides its compare method.

public int compare(T element1, T element2)

Returns a negative value if element1 is less than element2, returns a positive value if element1 is greater than element2, and returns zero if they are equal.

* The GeometricObject class (introduced in Section 13.2) does not implement the Comparable interface. To compare, you can define a comparator class.

```
import java.util.Comparator;

public class GeometricObjectComparator
    implements Comparator<GeometricObject>, java.io.Serializable {
  public int compare(GeometricObject o1, GeometricObject o2) {
    double area1 = o1.getArea();
    double area2 = o2.getArea();

    if (area1 < area2)
      return -1;
    else if (area1 == area2)
      return 0;
    else
      return 1;
  }
}
```

```
import java.util.Comparator;

public class TestComparator {
  public static void main(String[] args) {
  GeometricObject g1 = new Rectangle(5, 5);
  GeometricObject g2 = new Circle(5);

  GeometricObject g =
    max(g1, g2, new GeometricObjectComparator());

  System.out.println("The area of the larger object is " +
    g.getArea());
 }

 public static GeometricObject max(GeometricObject g1,
    GeometricObject g2, Comparator<GeometricObject> c) {
  if (c.compare(g1, g2) > 0)
    return g1;
  else
    return g2;
 }
}
```

```
The area of the larger object is 78.53981633974483
```

Comparable vs. Comparator

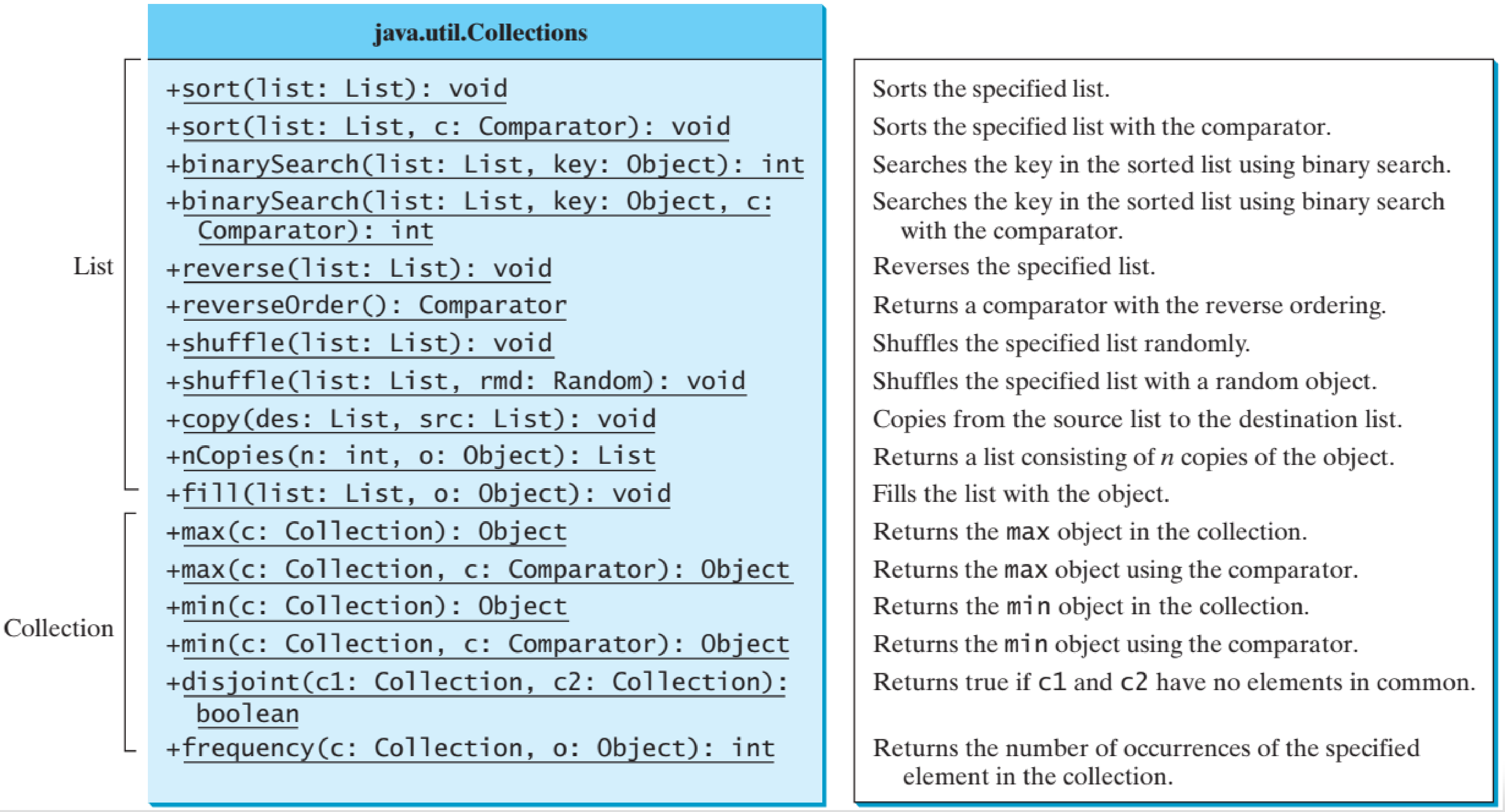
* Comparable is used to compare the objects of the class that implement Comparable.

* Comparator can be used to compare the objects of a class that doesn't implement Comparable.

- Comparing elements using the Comparable interface is referred to as comparing using natural order,
- and comparing elements using the Comparator interface is referred to as comparing using comparator.

Static Methods for Lists and Collections

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



	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.

Static Methods for Lists and Collections

* sort list

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

* ascending order vs. descending order

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");
Collections.sort(list, Collections.reverseOrder());
System.out.println(list);   //    output:  [yellow, red, green, blue].
```

* binarySearch

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

Static Methods for Lists and Collections

* reverse

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

* shuffle

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

* copy

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

Static Methods for Lists and Collections

* nCopies

```
      List<GregorianCalendar> list1 = Collections.nCopies
   (5, new GregorianCalendar(2005, 0, 1));
```

* fill

```
   List<String> list = Arrays.asList("red", "green", "blue");
   Collections.fill(list, "black");
   System.out.println(list);    // output: [black, black, black].
```

* max and min methods

```
 Collection<String> collection = Arrays.asList("red", "green", "blue");
 System.out.println(Collections.max(collection));
 System.out.println(Collections.min(collection));
```

Static Methods for Lists and Collections

* disjoint method

```
Collection<String> collection1 = Arrays.asList("red", "cyan");
Collection<String> collection2 = Arrays.asList("red", "blue");
Collection<String> collection3 = Arrays.asList("pink", "tan");
System.out.println(Collections.disjoint(collection1, collection2));
                                              //output: true
System.out.println(Collections.disjoint(collection1, collection3));
                                              // output: false
```

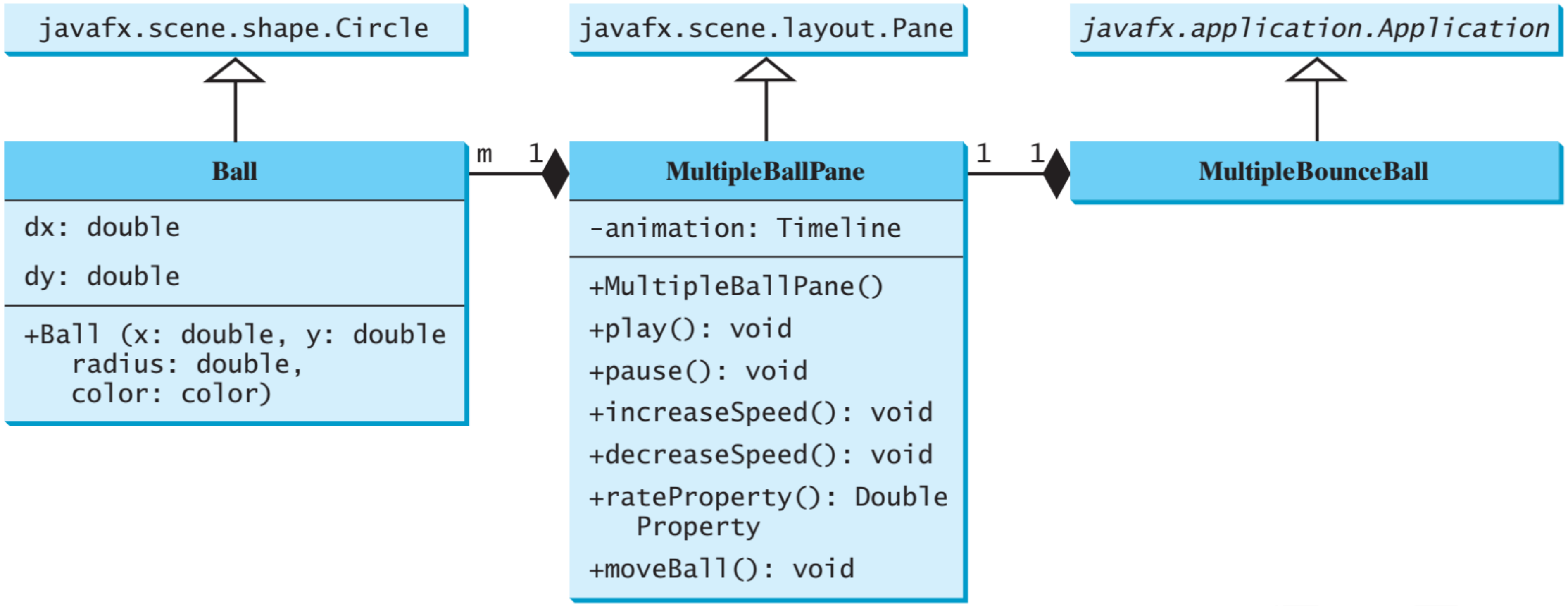
* frequency method

```
Collection<String> collection = Arrays.asList("red", "cyan", "red");
System.out.println(Collections.frequency(collection, "red"));
                                                // output: 2
```

Case Study: Multiple Bouncing Balls

* This section presents a program that displays multiple bouncing balls.

* You can use two buttons to suspend and resume the movement of the balls, a scroll bar to control the ball speed, and the + or - button add or remove a ball.



The Vector and Stack Classes

* The Java Collections Framework was introduced with Java 2.

* Several data structures were supported prior to Java 2.

- Among them are the Vector class and the Stack class.
- These classes were redesigned to fit into the Java Collections Framework,
- but their old-style methods are retained for compatibility.
- This section introduces the Vector class and the Stack class.

The Vector Class

* Vector is the same as ArrayList, except that Vector contains the synchronized methods for accessing and modifying the vector.

- None of the new collection data structures introduced so far are synchronized.

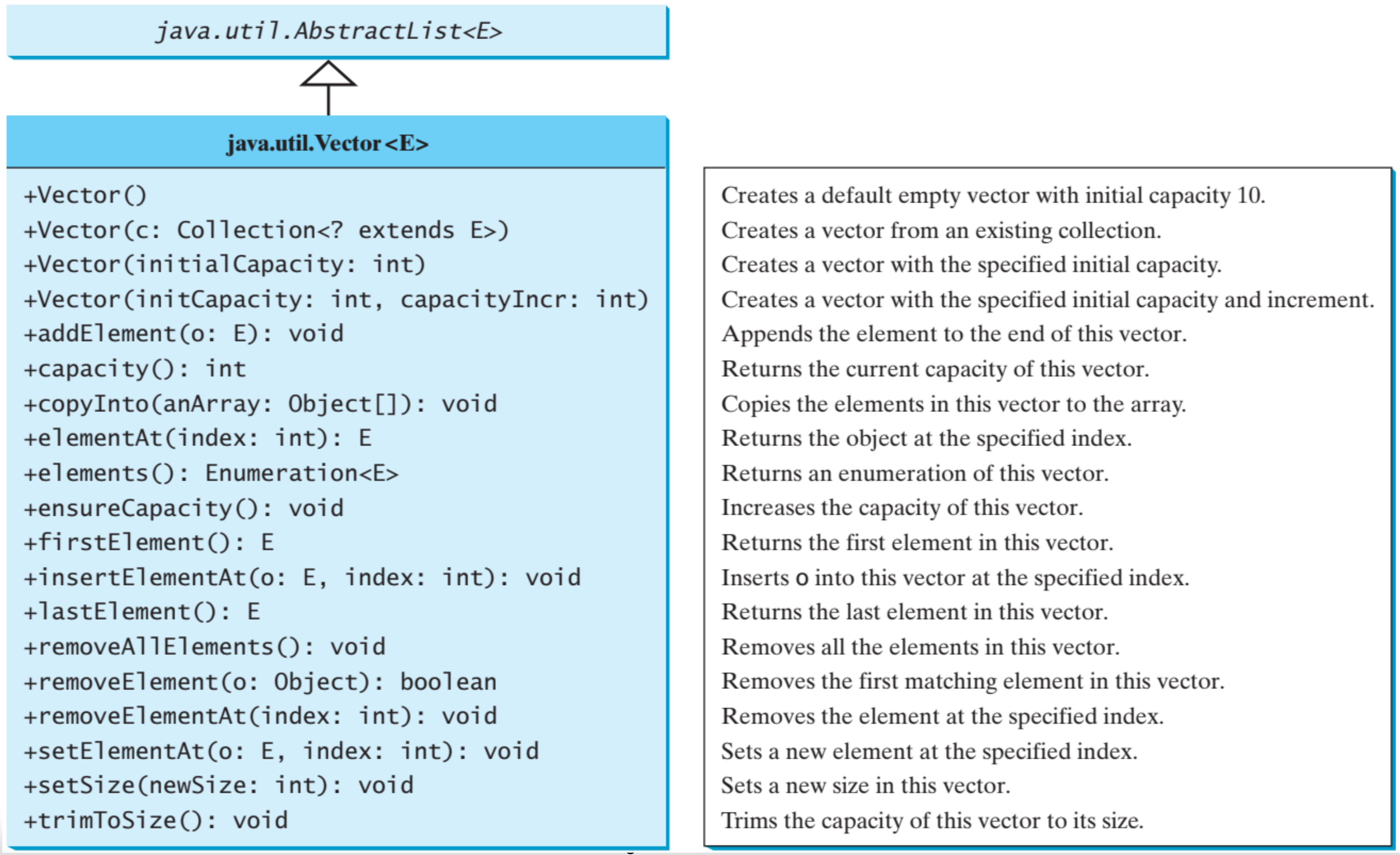
- Synchronized methods can prevent data corruption when a vector is accessed and modified by two or more threads concurrently.

- If synchronization is required, you can use the synchronized versions of the collection classes.

- For the many applications that do not require synchronization, using ArrayList is more efficient than using Vector.

The Vector Class, cont.

The Vector class extends the AbstractList class. It also has the methods contained in the original Vector class defined prior to Java 2.



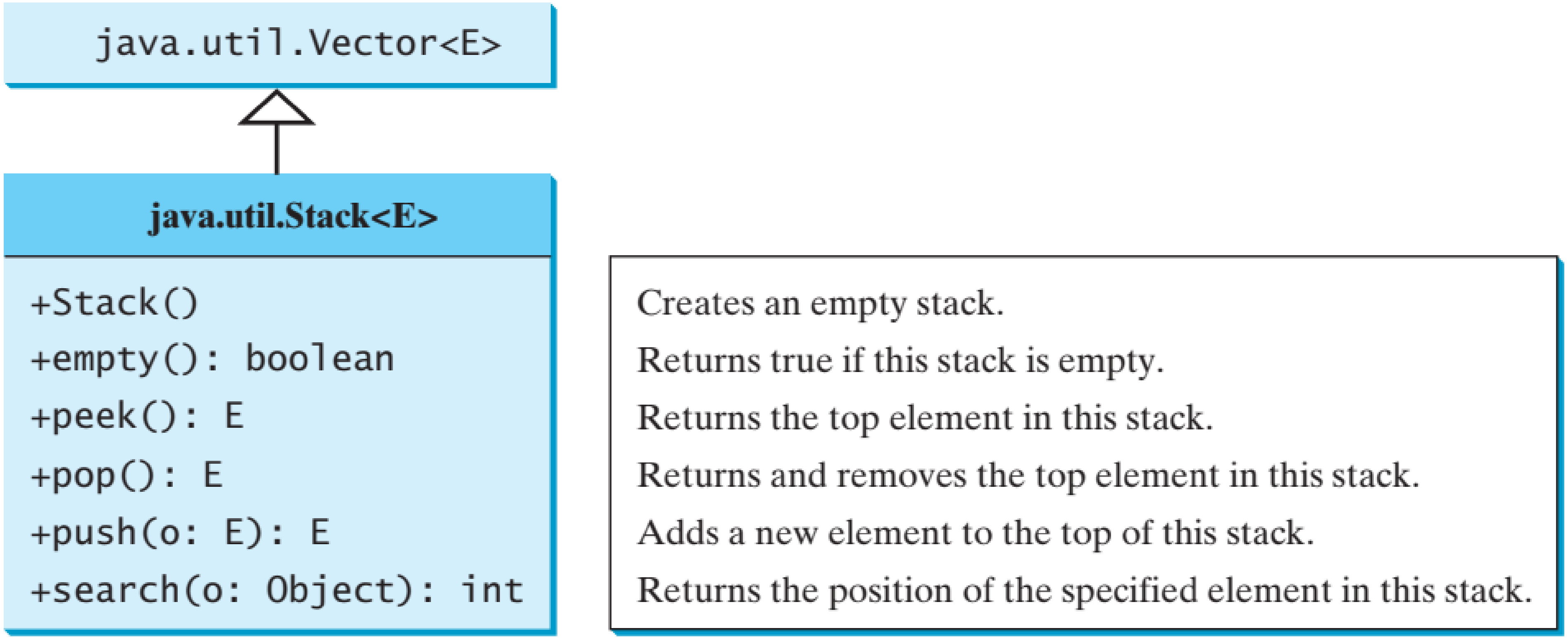
+Vector()	Creates a default empty vector with initial capacity 10.
+Vector(c: Collection<? extends E>)	Creates a vector from an existing collection.
+Vector(initialCapacity: int)	Creates a vector with the specified initial capacity.
+Vector(initCapacity: int, capacityIncr: int)	Creates a vector with 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.

The Stack Class

* In the Java Collections Framework, Stack is implemented as an extension of Vector.

* The Stack class represents a last-in-first-out stack of objects.

  - The elements are accessed only from the top of the stack.
  - You can retrieve, insert, or remove an element from the top of the stack.



`+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.

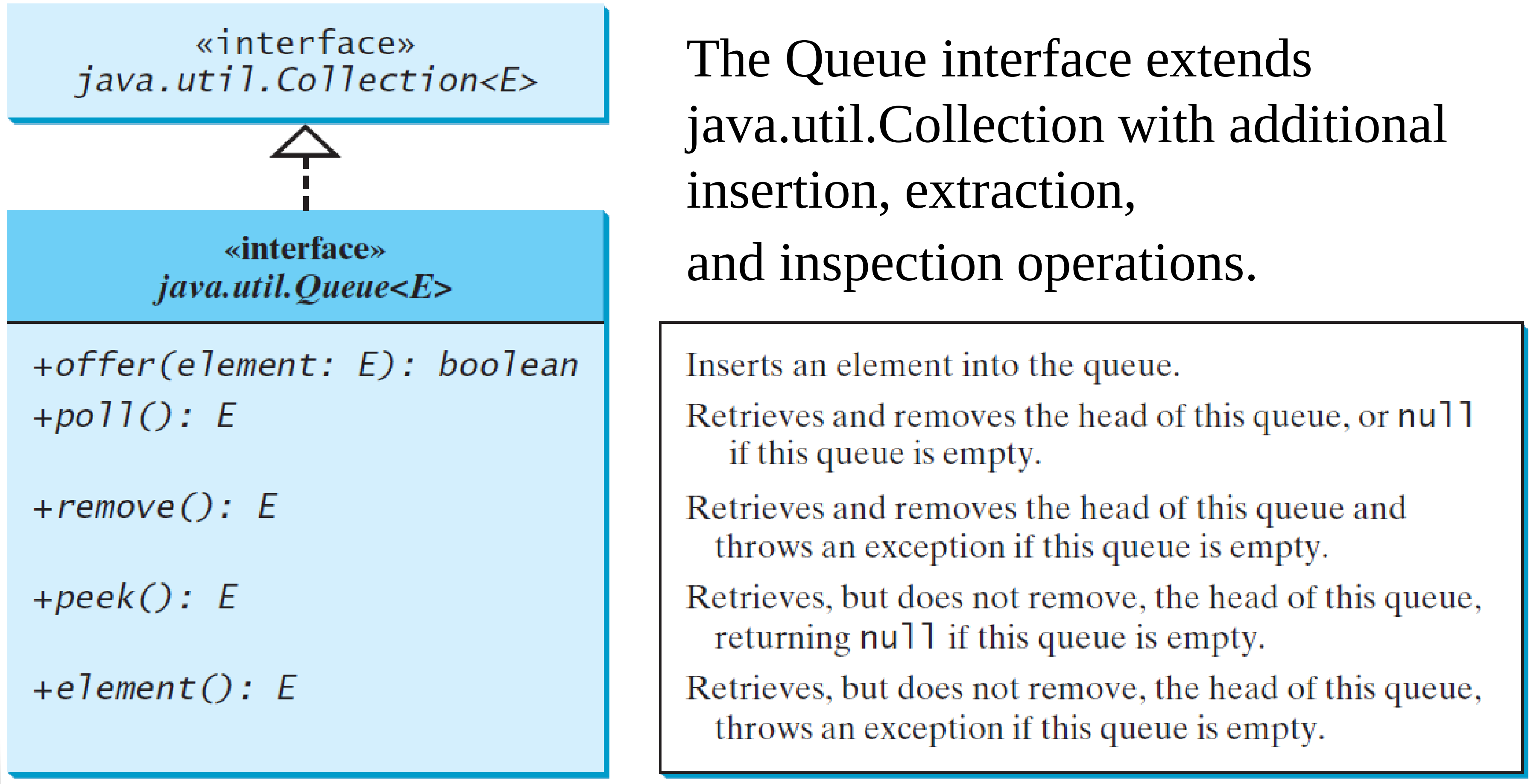
Queues and Priority Queues

* 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.

* In a priority queue, elements are assigned priorities.

- When accessing elements, the element with the highest priority is removed first.



+offer(element: E): boolean	Inserts an element into the queue.
+poll(): E	Retrieves and removes the head of this queue, or null if this queue is empty.
+remove(): E	Retrieves and removes the head of this queue and throws an exception if this queue is empty.
+peek(): E	Retrieves, but does not remove, the head of this queue, returning null if this queue is empty.
+element(): E	Retrieves, but does not remove, the head of this queue, throws an exception if this queue is empty.

The Queue interface extends java.util.Collection with additional insertion, extraction,

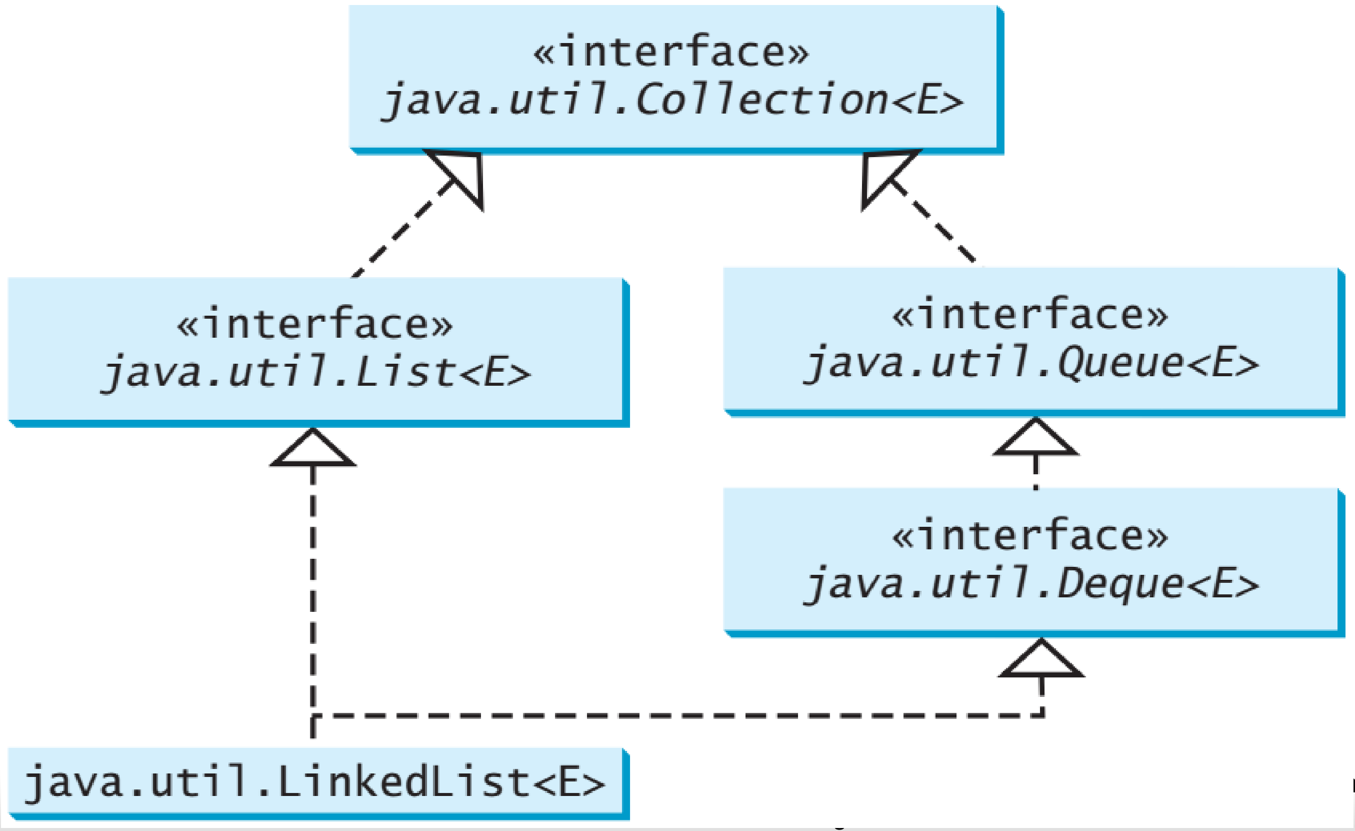
and inspection operations.

Deque and LinkedList

* The LinkedList class implements the Deque (double-ended queue) interface, which extends the Queue interface.

  - Therefore, you can use LinkedList to create a queue.

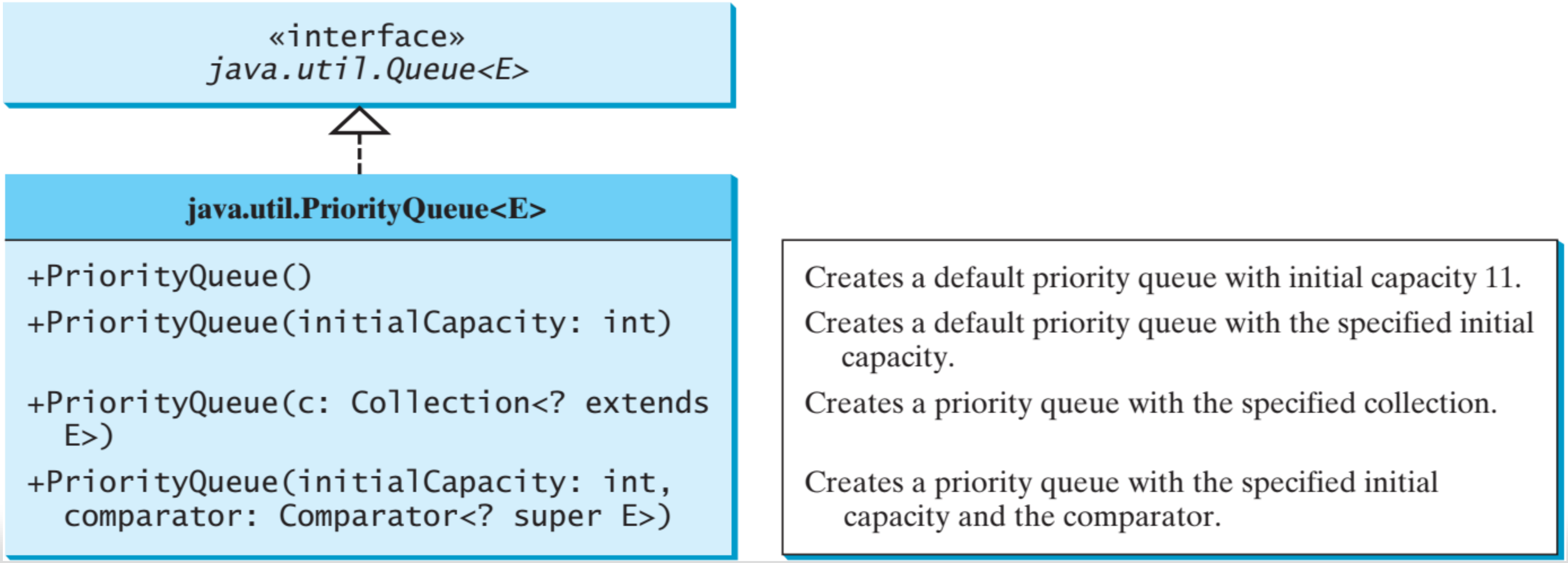
  - LinkedList is ideal for queue operations because it is efficient for inserting and removing elements from both ends of a list



The PriorityQueue Class

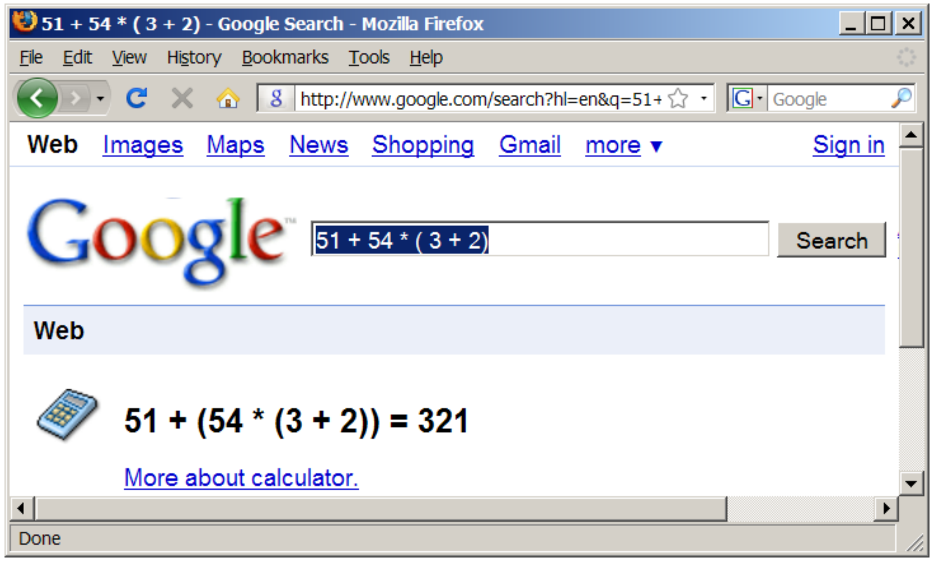
* The PriorityQueue class implements a priority queue.

- 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 same highest priority, the tie is broken arbitrarily.
- You can also specify an ordering using Comparator in the constructor PriorityQueue(initialCapacity, comparator).

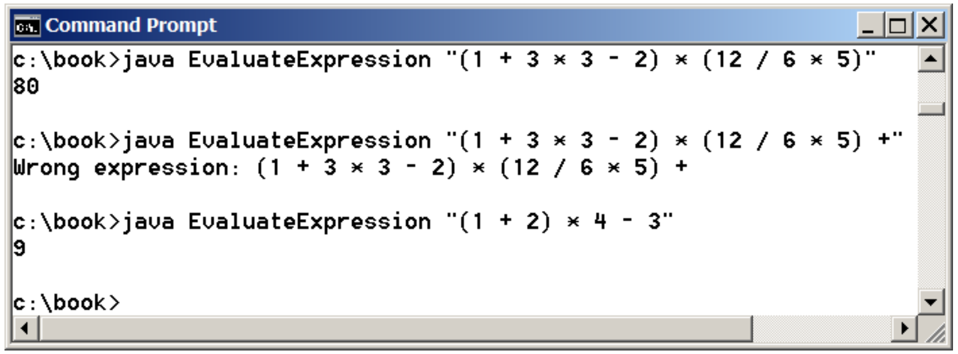

«interface»
java.util.Queue<E>
java.util.PriorityQueue<E>
+PriorityQueue()
+PriorityQueue(initialCapacity: int)
+PriorityQueue(c: Collection<? extends E>)
+PriorityQueue(initialCapacity: int, comparator: Comparator<? super E>)
Creates a default priority queue with initial capacity 11.
Creates a default 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.

Case Study: Evaluating Expressions

Stacks can be used to evaluate expressions.


51 + 54 * (3 + 2) - Google Search - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.google.com/search?hl=en&q=51+
Web Images Maps News Shopping Gmail more ▾ Sign in
51 + 54 * (3 + 2)
Search
Web
More about calculator.
Done

51 + (54 * (3 + 2)) = 321



```
c:\book>java EvaluateExpression "(1 + 3 * 3 - 2) * (12 / 6 * 5)"
80

c:\book>java EvaluateExpression "(1 + 3 * 3 - 2) * (12 / 6 * 5) +"
Wrong expression: (1 + 3 * 3 - 2) * (12 / 6 * 5) +

c:\book>java EvaluateExpression "(1 + 2) * 4 - 3"
9

c:\book>
```

* This section presents a program that evaluates a compound expression with multiple operators and parentheses (e.g., (15 + 2) * 34 – 2).

* For simplicity, assume that the operands are integers and the operators are of four types: +, -, *, and /.

Method

* The problem can be solved using two stacks, named operandStack and operatorStack, for storing operands and operators, respectively.

- Operands and operators are pushed into the stacks before they are processed.

- When an operator is processed, it is popped from operatorStack and applied to the first two operands from operandStack (the two operands are popped from operandStack).

- The resultant value is pushed back to operandStack.

Algorithm

Phase 1: Scanning the expression

The program scans the expression from left to right to extract operands, operators, and the parentheses.

1.1. If the extracted item is an operand, push it to **operandStack**.

1.2. If the extracted item is a **+** or **-** operator, process all the operators at the top of **operatorStack** and push the extracted operator to **operatorStack**.

1.3. If the extracted item is a ***** or **/** operator, process the ***** or **/** operators at the top of **operatorStack** and push the extracted operator to **operatorStack**.

1.4. If the extracted item is a **(** symbol, push it to **operatorStack**.

1.5. If the extracted item is a **)** symbol, repeatedly process the operators from the top of **operatorStack** until seeing the **(** symbol on the stack.

Phase 2: Clearing the stack

Repeatedly process the operators from the top of **operatorStack** until **operatorStack** is empty.

Example

Expression	*Scan*	*Action*	operandStack	operatorStack
$\underset{\uparrow}{(}1 + 2)*4 - 3$	(	Phase 1.4		(
$(\underset{\uparrow}{1} + 2)*4 - 3$	1	Phase 1.1	1	(
$(1 \underset{\uparrow}{+} 2)*4 - 3$	+	Phase 1.2	1	$\begin{array}{c}+\\(\end{array}$
$(1 + \underset{\uparrow}{2})*4 - 3$	2	Phase 1.1	$\begin{array}{c}2\\1\end{array}$	(
$(1 + 2\underset{\uparrow}{)}*4 - 3$	)	Phase 1.5	3	
$(1 + 2)\underset{\uparrow}{*}4 - 3$	*	Phase 1.3	3	*
$(1 + 2)*\underset{\uparrow}{4} - 3$	4	Phase 1.1	$\begin{array}{c}4\\3\end{array}$	*
$(1 + 2)*4 \underset{\uparrow}{-} 3$	–	Phase 1.2	12	–
$(1 + 2)*4 - \underset{\uparrow}{3}$	3	Phase 1.1	$\begin{array}{c}3\\12\end{array}$	–
$(1 + 2)*4 - 3\ \underset{\uparrow}{}$	none	Phase 2	9	