

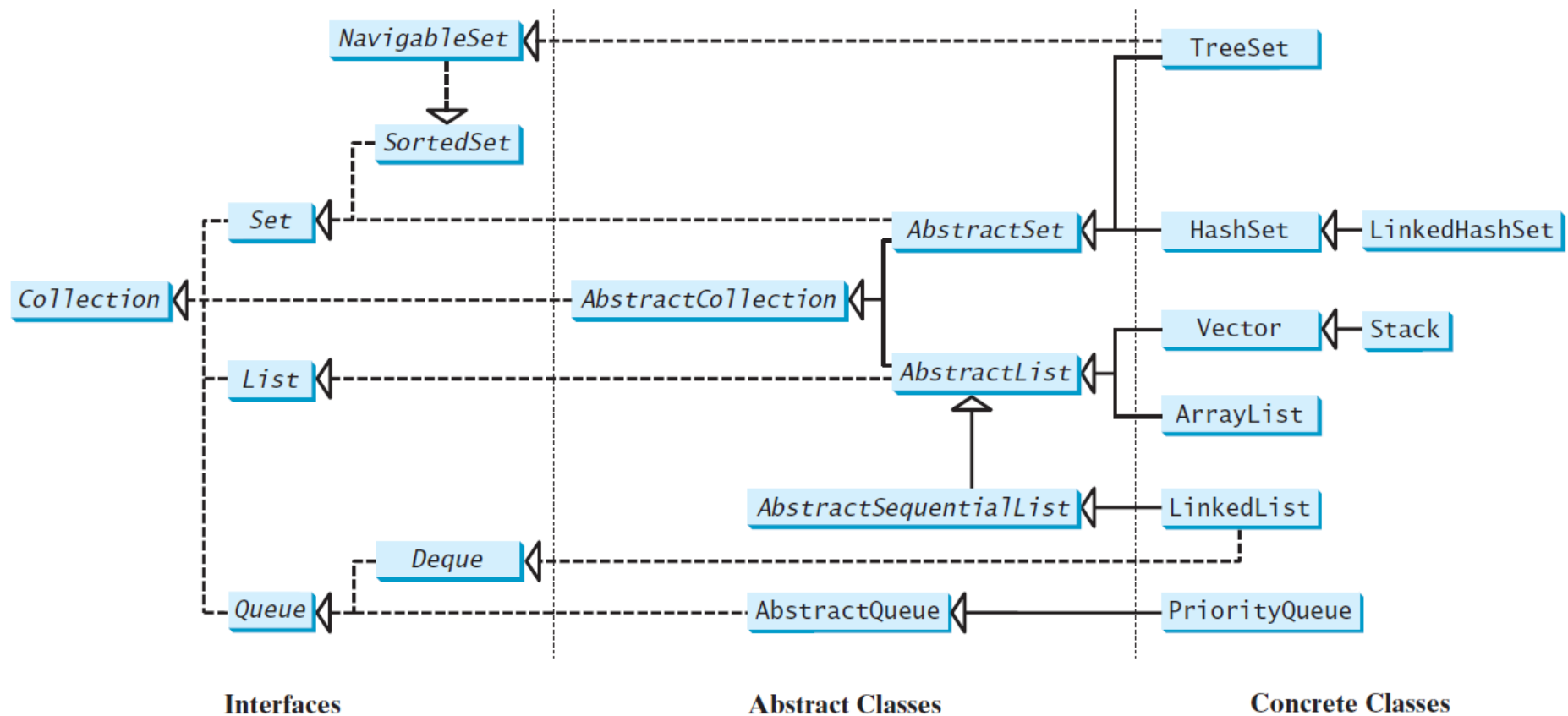
# Chapter 20 Collection Framework: Sets , Lists and Maps

# Java Collection Framework hierarchy

*A collection* is a container object, in fact Interface, that holds a group of objects, often referred to as *elements*. The Java Collections Framework supports three types of collections, named *lists*, *sets*, and *maps*.

# Java Collection Framework hierarchy, cont.

Set and List are subinterfaces of Collection.



# The Collection Interface

Returns an iterator for the elements in this collection.

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

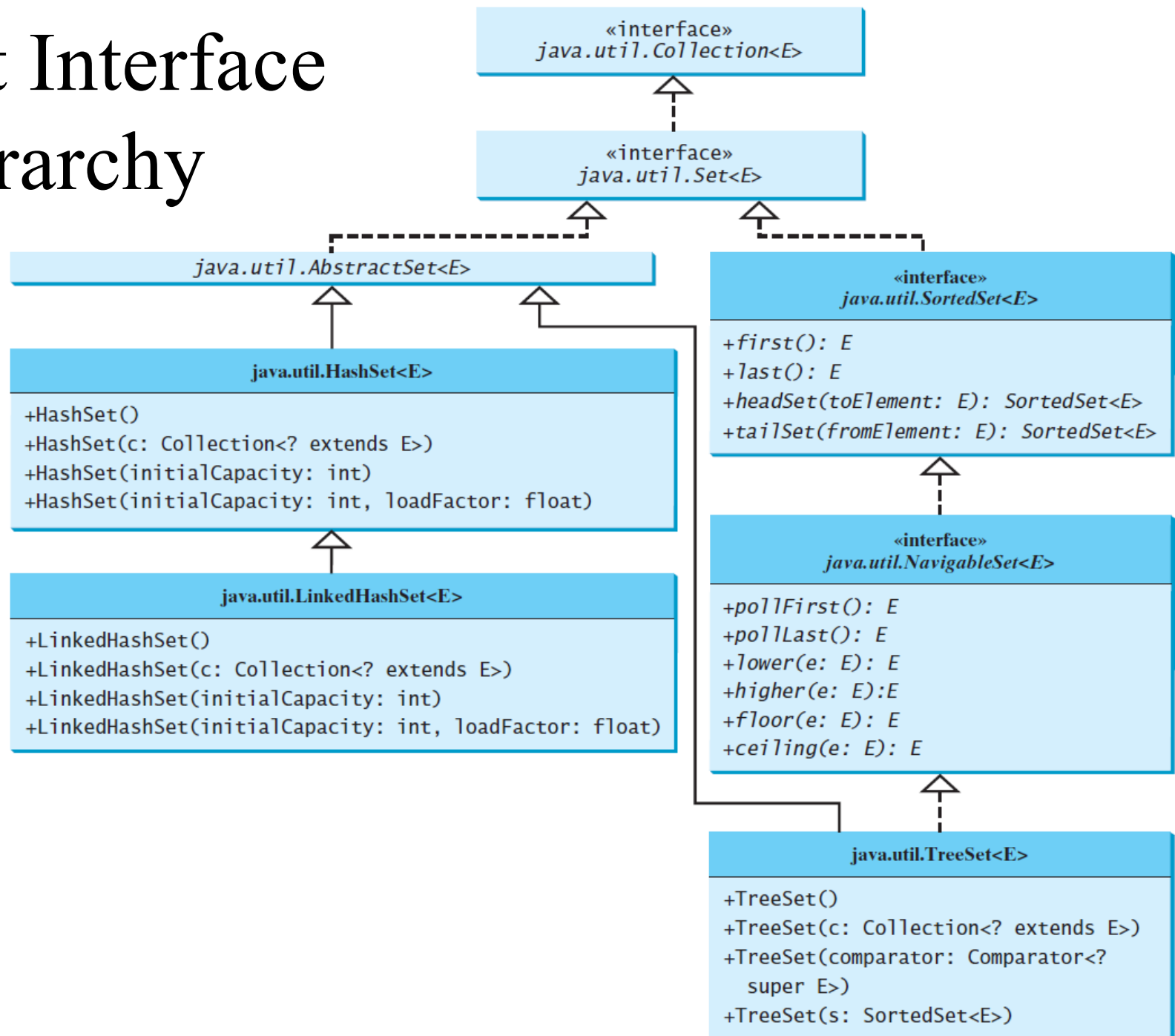
Adds a new element `o` to this collection.  
Adds all the elements in the collection `c` to this collection.  
Removes all the elements from this collection.  
Returns true if this collection contains the element `o`.  
Returns true if this collection contains all the elements in `c`.  
Returns true 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.

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 Set Interface

The Set interface extends the Collection interface. It does not introduce new methods or constants, but it stipulates that an instance of Set contains no duplicate elements. The concrete classes that implement Set must ensure that no duplicate elements can be added to the set. That is no two elements `e1` and `e2` can be in the set such that `e1.equals(e2)` is true.

# The Set Interface Hierarchy



# The AbstractSet Class

The AbstractSet class is a convenience class that extends AbstractCollection and implements Set. The AbstractSet class provides concrete implementations for the equals method and the hashCode method. The hash code of a set is the sum of the hash code of all the elements in the set. Since the size method and iterator method are not implemented in the AbstractSet class, AbstractSet is an abstract class.

# The HashSet Class

The HashSet class is a concrete class that implements Set. It can be used to store duplicate-free elements. For efficiency, objects added to a hash set need to implement the hashCode method in a manner that properly disperses the hash code.



# Example: Using HashSet and Iterator

This example creates a hash set filled with strings, and uses an iterator to traverse the elements in the list.

TestHashSet

# TIP: for-each loop

You can simplify the code in Lines 21-26 using a JDK 1.5 enhanced for loop without using an iterator, as follows:

```
for (Object element: set)
    System.out.print(element.toString() + " ");
```

# Example: Using LinkedHashSet

This example creates a hash set filled with Strings, and uses an iterator to traverse the elements in the list.

TestLinkedHashSet

# The SortedSet Interface and the TreeSet Class

SortedSet is a subinterface of Set, which guarantees that the elements in the set are sorted. TreeSet is a concrete class that implements the SortedSet interface. You can use an iterator to traverse the elements in the sorted order. The elements can be sorted in two ways.

# The SortedSet Interface and the TreeSet Class, cont.

One way is to use the Comparable interface.

The other way is to specify a comparator for the elements in the set if the class for the elements does not implement the Comparable interface, or you don't want to use the compareTo method in the class that implements the Comparable interface. This approach is referred to as *order by comparator*.

# Example: Using TreeSet to Sort Elements in a Set

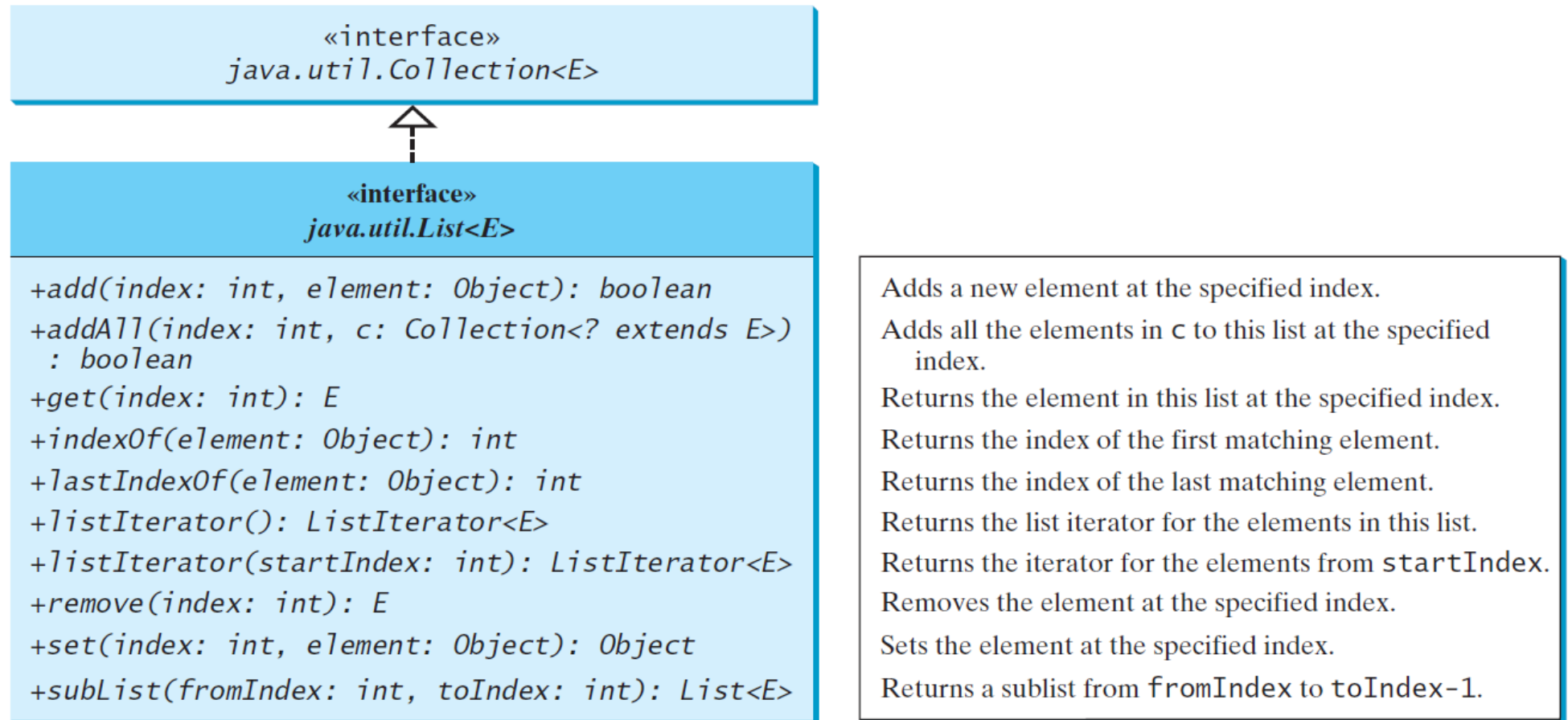
This example creates a hash set filled with strings, and then creates a tree set for the same strings. The strings are sorted in the tree set using the compareTo method in the Comparable interface. The example also creates a tree set of geometric objects. The geometric objects are sorted using the compare method in the Comparator interface.

TestTreeSet

# The List Interface

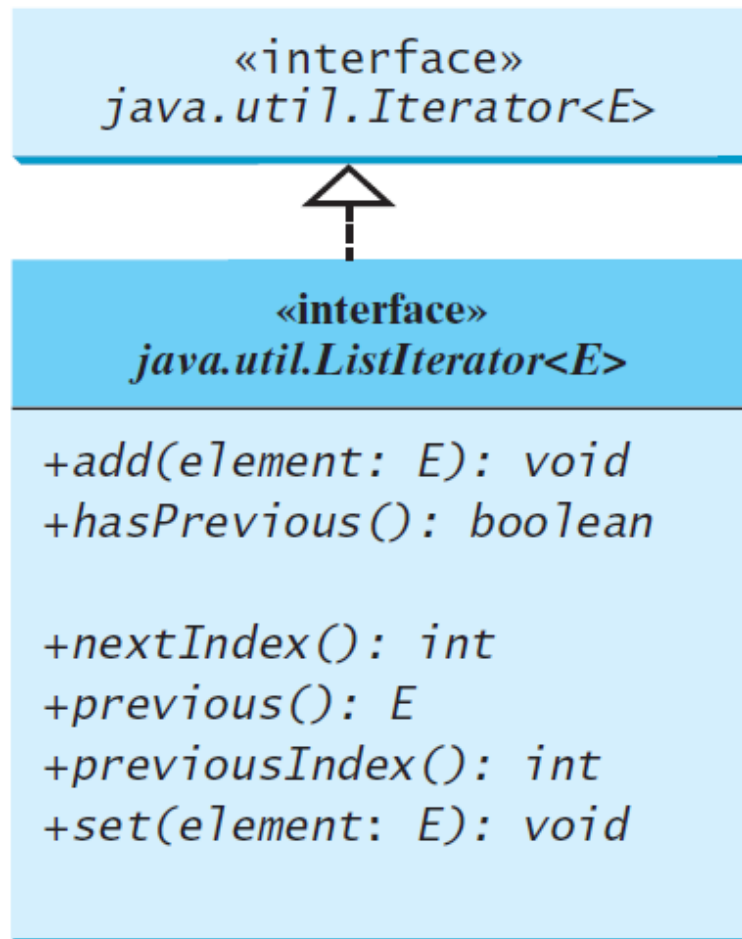
A list stores elements in a sequential order, and allows the user to specify where the element is stored. The user can access the elements by index.

# The List Interface, cont.





# The List Iterator



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

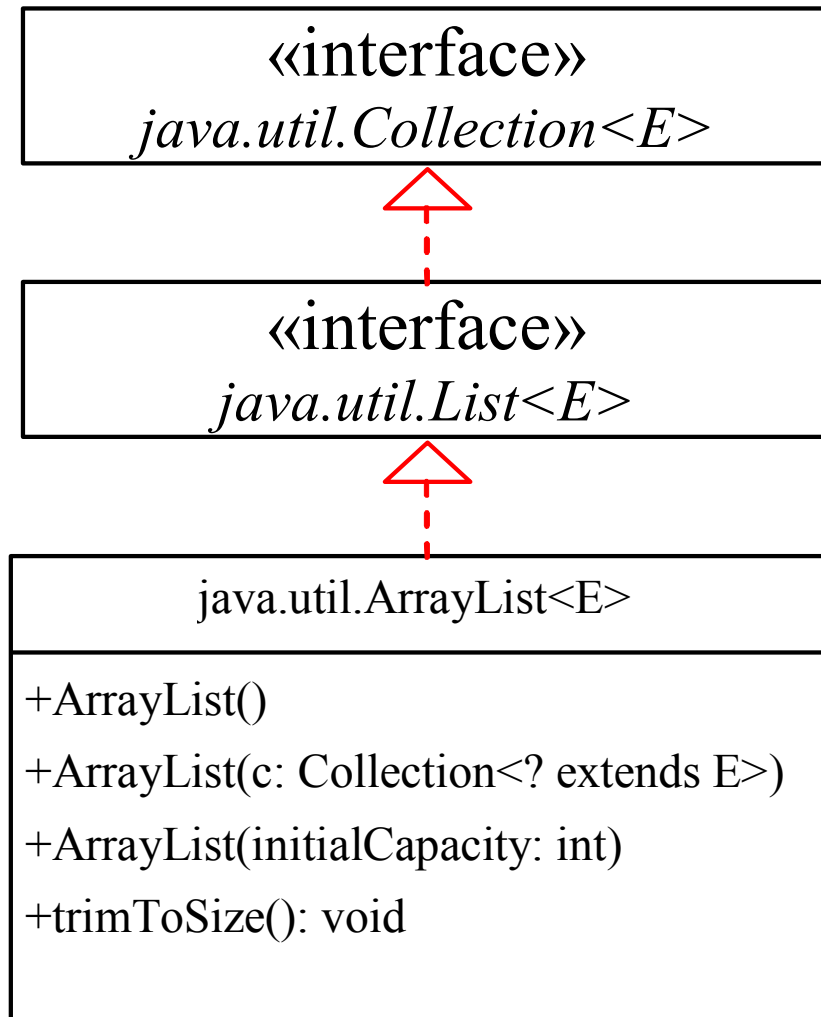
The ArrayList class and the LinkedList class are concrete implementations of the List interface.

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



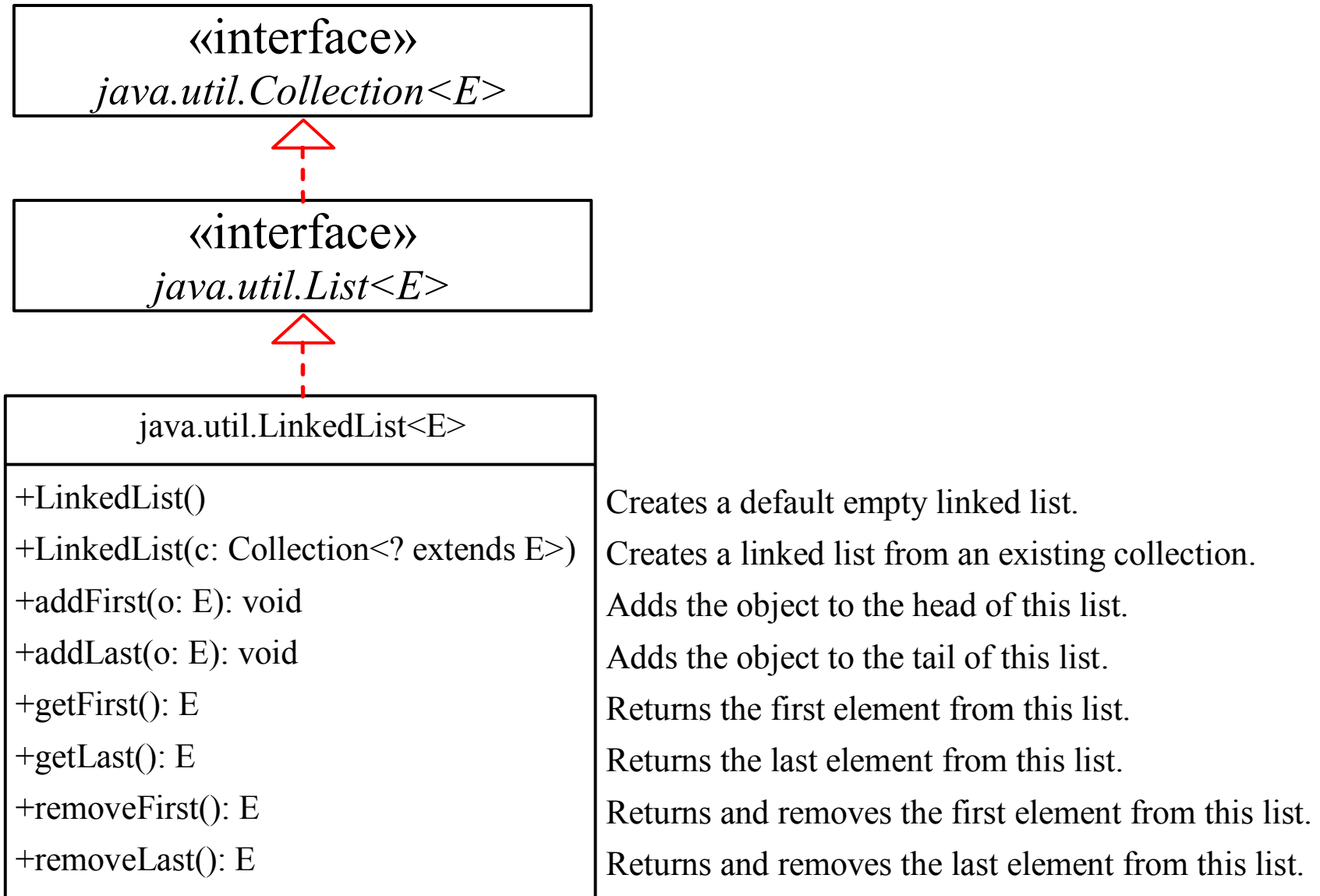
Creates an empty list with the default initial capacity.

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



# 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.

TestArrayAndLinkedList

# The Comparator Interface

Sometimes you want to compare the elements of different types. The elements may not be instances of Comparable or are not comparable. You can define a comparator to compare these elements. To do so, define a class that implements the `java.util.Comparator` interface.

The Comparator interface has two methods, `compare` and `equals`.

# The Comparator Interface

```
public int compare(Object element1, Object element2)
```

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

[GeometricObjectComparator](#)

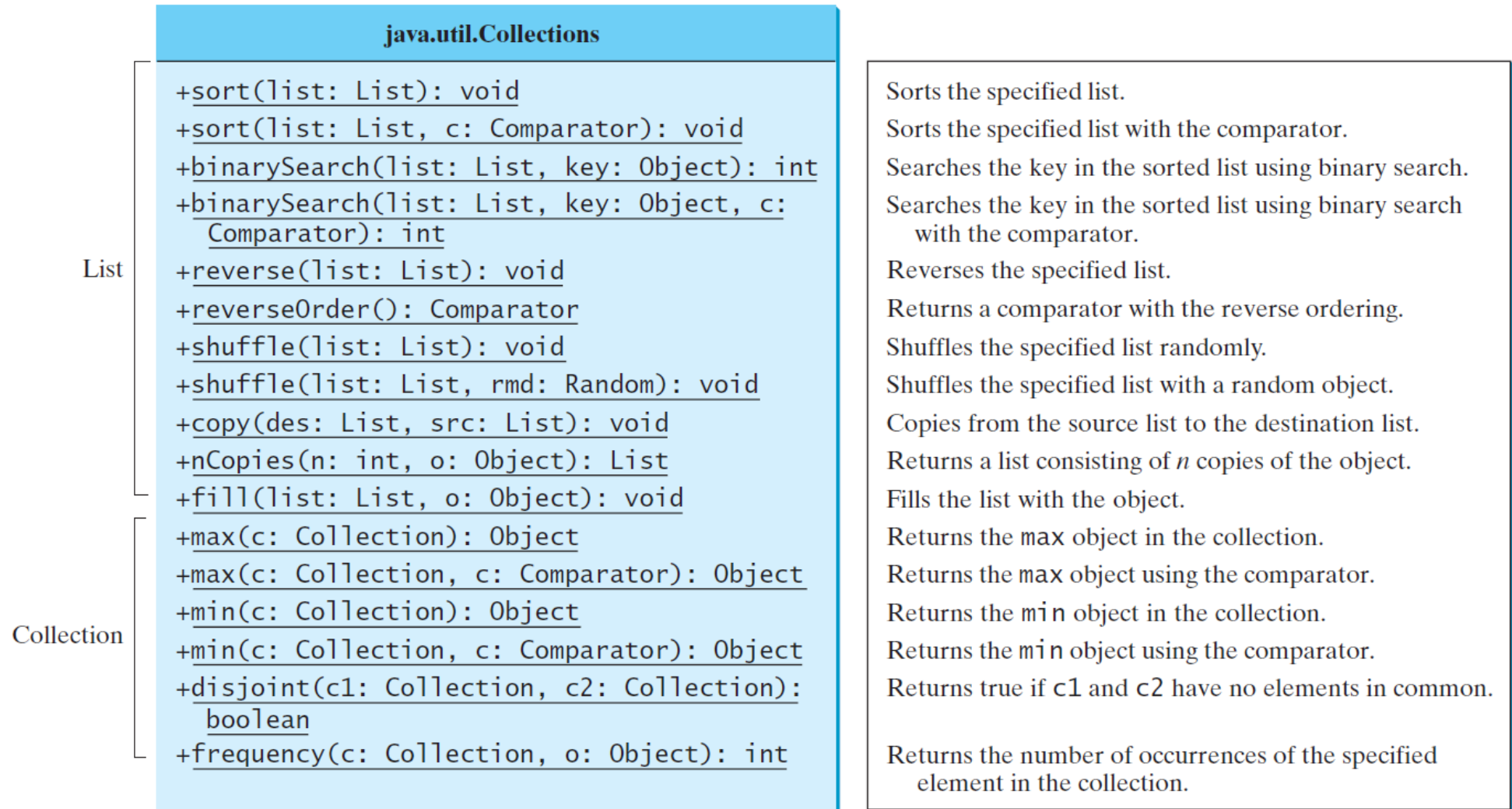
[TestTreeSetWithComparator](#)

# 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.



# The Collections Class UML Diagram



# 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.

# The Vector Class

In Java 2, 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. If synchronization is required, you can use the synchronized versions of the collection classes.

# The Vector Class, cont.

*java.util.AbstractList<E>*



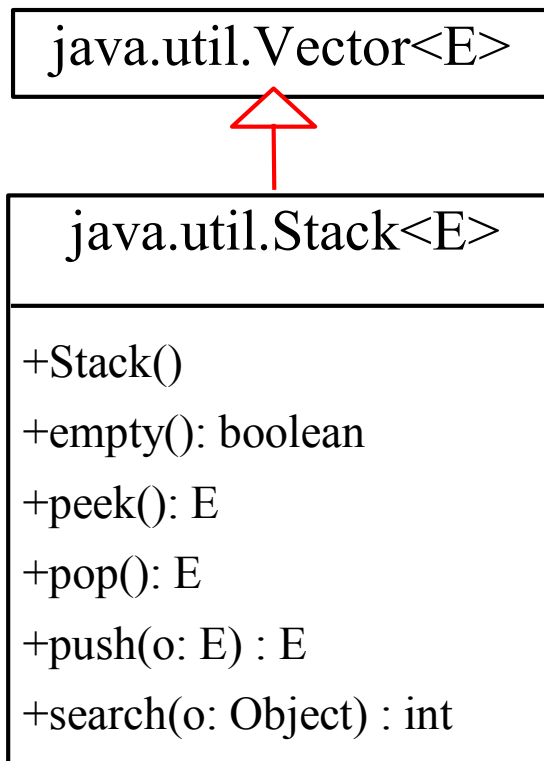
**java.util.Vector<E>**

```
+Vector()  
+Vector(c: Collection<? extends E>)  
+Vector(initialCapacity: int)  
+Vector(initCapacity: int, capacityIncr: int)  
+addElement(o: E): void  
+capacity(): int  
+copyInto(anArray: Object[]): void  
+elementAt(index: int): E  
+elements(): Enumeration<E>  
+ensureCapacity(): void  
+firstElement(): E  
+insertElementAt(o: E, index: int): void  
+lastElement(): E  
+removeAllElements(): void  
+removeElement(o: Object): boolean  
+removeElementAt(index: int): void  
+setElementAt(o: E, index: int): void  
+setSize(newSize: int): void  
+trimToSize(): void
```

Creates a default empty vector with initial capacity 10.  
Creates a vector from an existing collection.  
Creates a vector with the specified initial capacity.  
Creates a vector with the specified initial capacity and increment.  
Appends the element to the end of this vector.  
Returns the current capacity of this vector.  
Copies the elements in this vector to the array.  
Returns the object at the specified index.  
Returns an enumeration of this vector.  
Increases the capacity of this vector.  
Returns the first element in this vector.  
Inserts *o* into this vector at the specified index.  
Returns the last element in this vector.  
Removes all the elements in this vector.  
Removes the first matching element in this vector.  
Removes the element at the specified index.  
Sets a new element at the specified index.  
Sets a new size in this vector.  
Trims the capacity of this vector to its size.

# The Stack Class

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.



Creates an empty stack.

Returns true if this stack is empty.

Returns the top element in this stack.

Returns and removes the top element in this stack.

Adds a new element to the top of this stack.

Returns the position of the specified element in this stack.

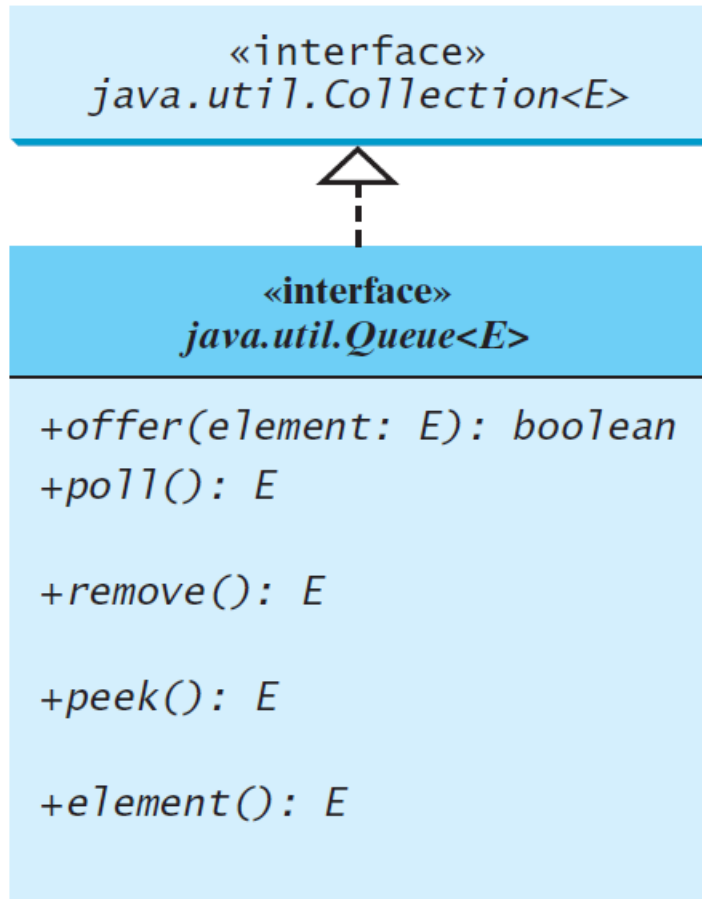
# Queues and Priority Queues

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.

# The Queue Interface



Inserts an element into the queue.

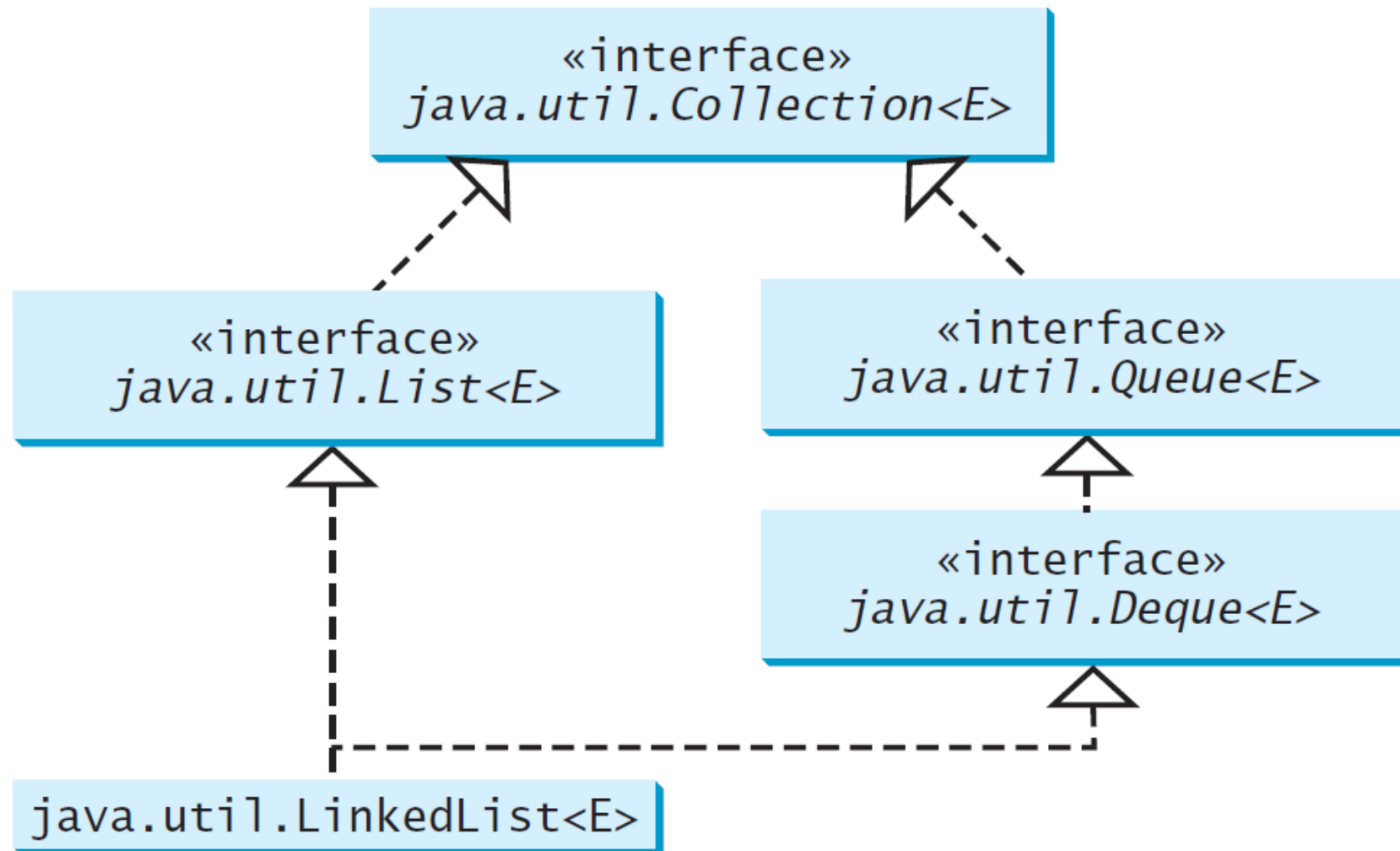
Retrieves and removes the head of this queue, or `null` if this queue is empty.

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.

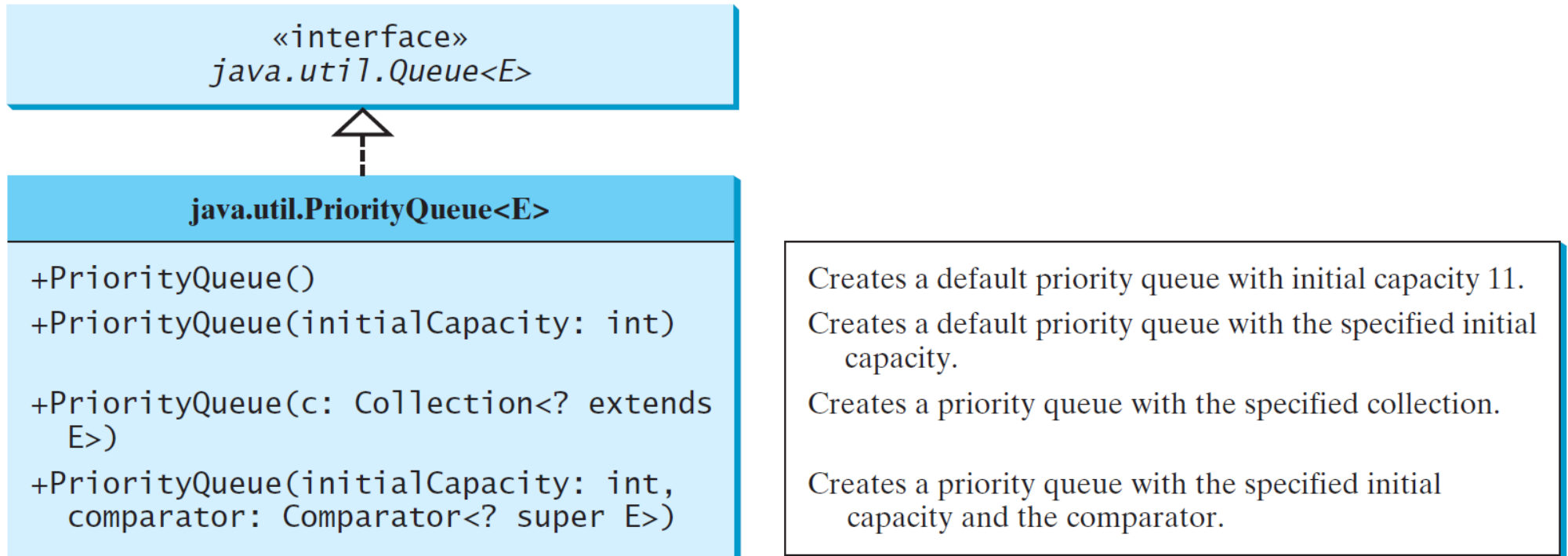
Retrieves, but does not remove, the head of this queue, throws an exception if this queue is empty.

# Using LinkedList for Queue





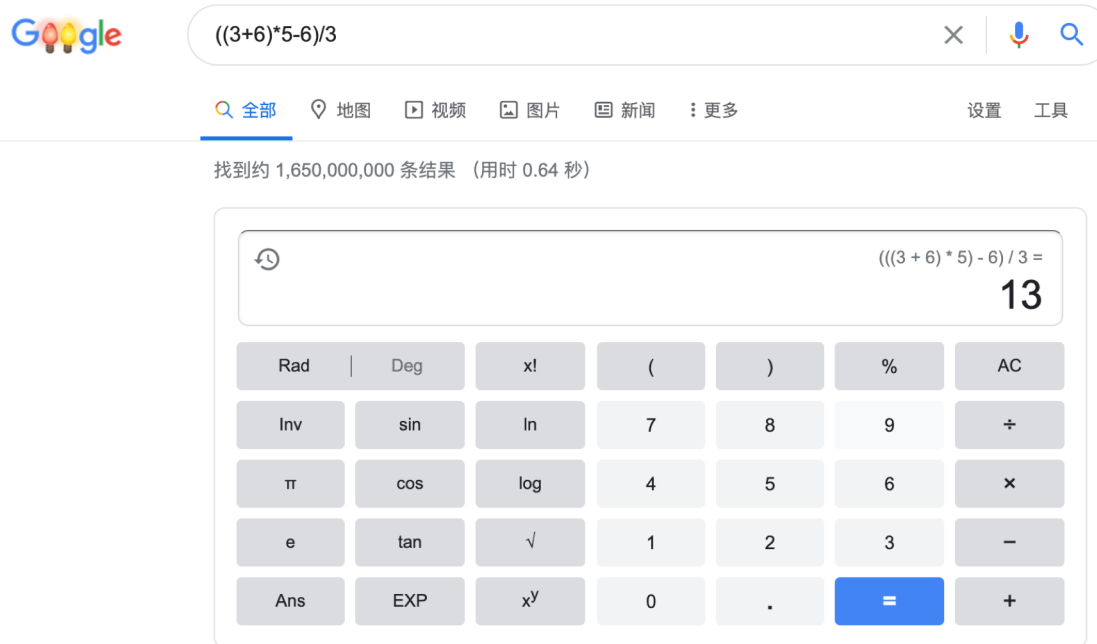
# The PriorityQueue Class



PriorityQueueDemo

# Case Study: Evaluating Expressions

Stacks can be used to evaluate expressions.



Evaluate Expression

```
(base) → Java_Course git:(master) x java ch20.EvaluateExpression "((5+6)*3-6)/3+7"
```

```
16
```

```
(base) → Java_Course git:(master) x java ch20.EvaluateExpression "((5+6/2)*12-(6+2))/4+11"
```

```
33
```

```
(base) → Java_Course git:(master) x █
```

# 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

<i>Expression</i>	<i>Scan</i>	<i>Action</i>	operandStack	operatorStack
(1 + 2)*4 - 3 ↑	(	Phase 1.4		(
(1 + 2)*4 - 3 ↑	1	Phase 1.1	1	(
(1 + 2)*4 - 3 ↑	+	Phase 1.2	1	+
(1 + 2)*4 - 3 ↑	2	Phase 1.1	2 1	(
(1 + 2)*4 - 3 ↑	)	Phase 1.5	3	
(1 + 2)*4 - 3 ↑	*	Phase 1.3	3	*
(1 + 2)*4 - 3 ↑	4	Phase 1.1	4 3	*
(1 + 2)*4 - 3 ↑	-	Phase 1.2	12	-
(1 + 2)*4 - 3 ↑	3	Phase 1.1	3 12	-
(1 + 2)*4 - 3 ↑	none	Phase 2	9	

# Performance of Sets and Lists

SetListPerformanceTest

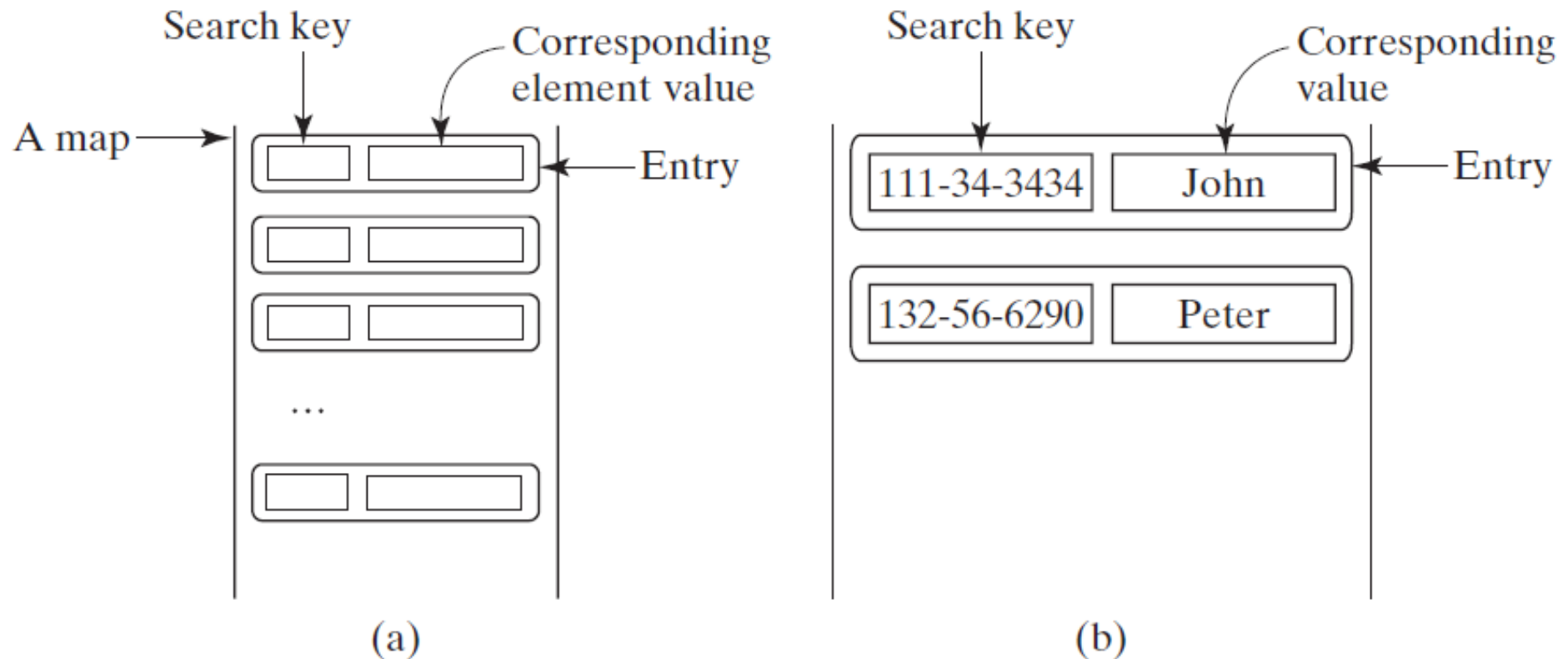
# Case Study: Counting Keywords

This section presents an application that counts the number of the keywords in a Java source file.

CountKeywords

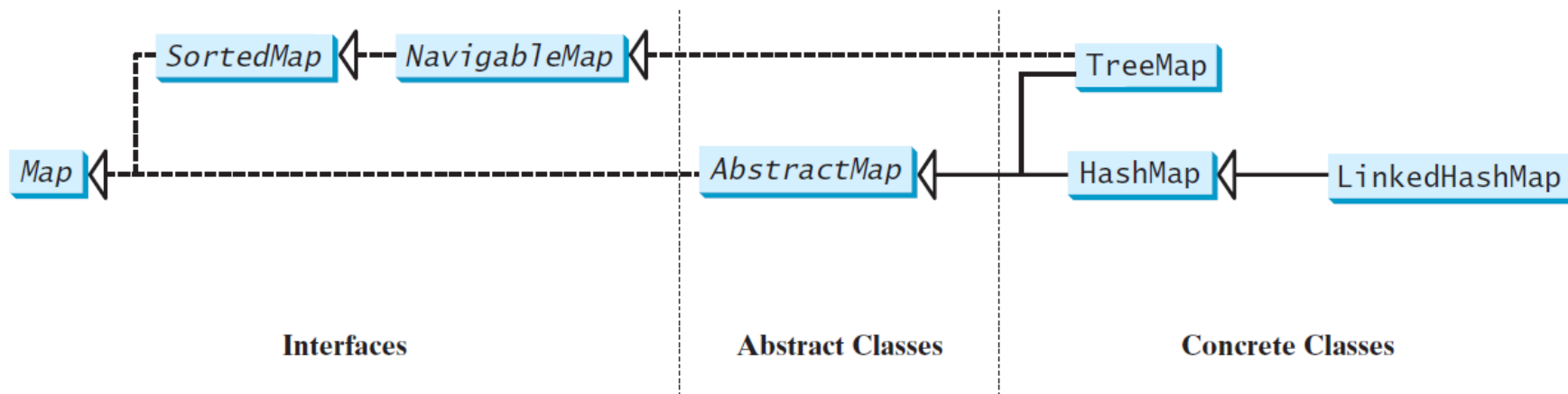
# The Map Interface

The Map interface maps keys to the elements. The keys are like indexes. In List, the indexes are integer. In Map, the keys can be any objects.



# Map Interface and Class Hierarchy

An instance of Map represents a group of objects, each of which is associated with a key. You can get the object from a map using a key, and you have to use a key to put the object into the map.





# The Map Interface UML Diagram

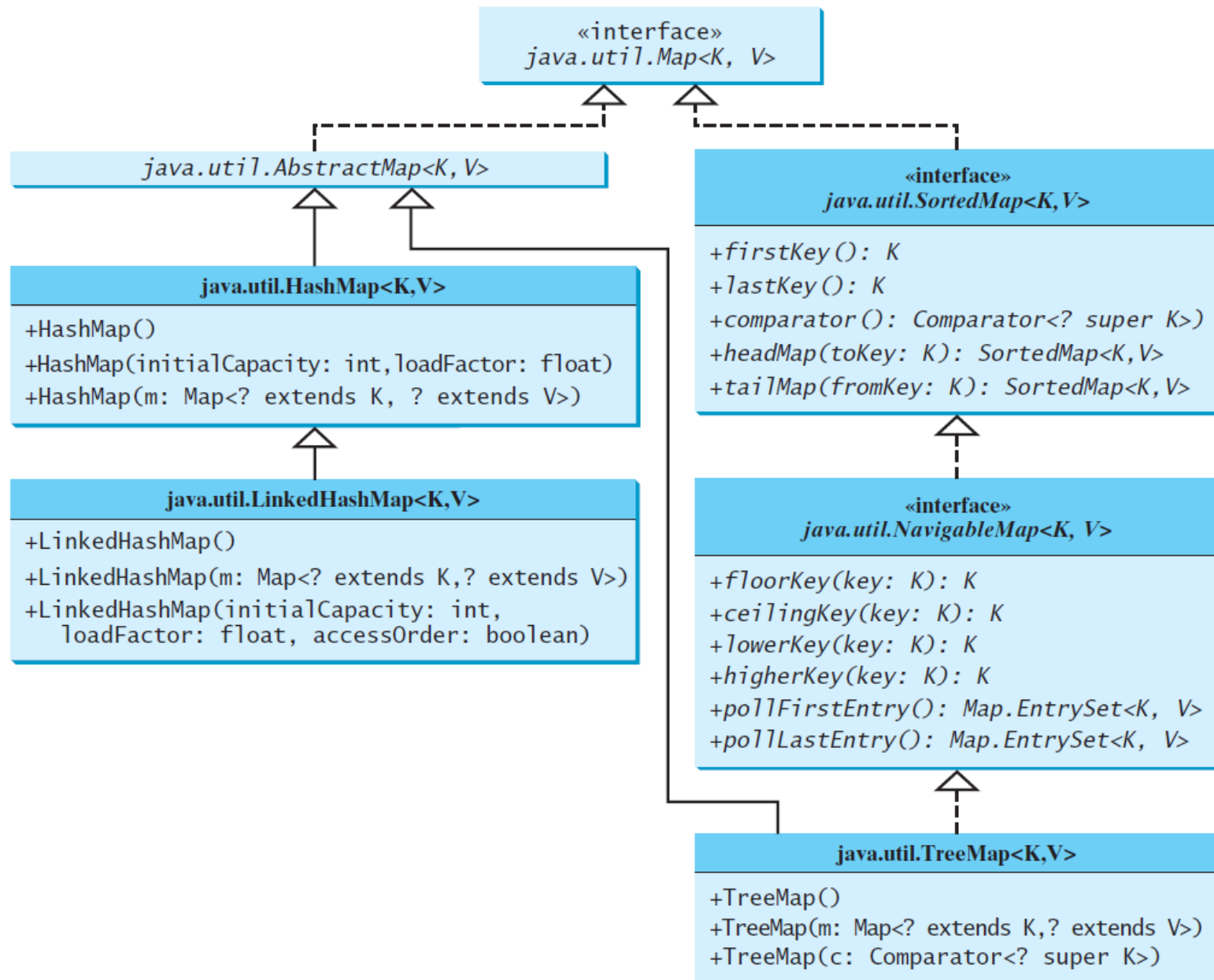
**«interface»**  
*java.util.Map<K, V>*

```
+clear(): void  
+containsKey(key: Object): boolean  
  
+containsValue(value: Object): boolean  
  
+entrySet(): Set<Map.Entry<K, V>>  
+get(key: Object): V  
+isEmpty(): boolean  
+keySet(): Set<K>  
+put(key: K, value: V): V  
+putAll(m: Map<? extends K, ? extends V>): void  
+remove(key: Object): V  
+size(): int  
+values(): Collection<V>
```

Removes all entries from this map.  
Returns true if this map contains an entry for the specified key.  
Returns true if this map maps one or more keys to the specified value.  
Returns a set consisting of the entries in this map.  
Returns the value for the specified key in this map.  
Returns true if this map contains no entries.  
Returns a set consisting of the keys in this map.  
Puts an entry into this map.  
Adds all the entries from m to this map.

Removes the entries for the specified key.  
Returns the number of entries in this map.  
Returns a collection consisting of the values in this map.

# Concrete Map Classes



# Entry

**«interface»**  
*java.util.Map.Entry<K, V>*

*+getKey(): K*  
*+getValue(): V*  
*+setValue(value: V): void*

Returns the key from this entry.

Returns the value from this entry.

Replaces the value in this entry with a new value.

# HashMap and TreeMap

The HashMap and TreeMap classes are two concrete implementations of the Map interface.

The HashMap class is efficient for locating a value, inserting a mapping, and deleting a mapping.

The TreeMap class, implementing SortedMap, is efficient for traversing the keys in a sorted order.

# LinkedHashMap

LinkedHashMap was introduced in JDK 1.4. It extends HashMap with a linked list implementation that supports an ordering of the entries in the map. The entries in a HashMap are not ordered, but the entries in a LinkedHashMap can be retrieved in the order in which they were inserted into the map (known as the insertion order), or the order in which they were last accessed, from least recently accessed to most recently (access order). The no-arg constructor constructs a LinkedHashMap with the insertion order. To construct a LinkedHashMap with the access order, use the LinkedHashMap(initialCapacity, loadFactor, true).

# Example: Using HashMap and TreeMap

This example creates a hash map that maps borrowers to mortgages. The program first creates a hash map with the borrower's name as its key and mortgage as its value. The program then creates a tree map from the hash map, and displays the mappings in ascending order of the keys.

TestMap

# Case Study: Counting the Occurrences of Words in a Text

This program counts the occurrences of words in a text and displays the words and their occurrences in ascending order of the words. The program uses a hash map to store a pair consisting of a word and its count. For each word, check whether it is already a key in the map. If not, add the key and value 1 to the map. Otherwise, increase the value for the word (key) by 1 in the map. To sort the map, convert it to a tree map.

CountOccurrenceOfWords

# The Singleton and Unmodifiable Collections

## `java.util.Collections`

```
+singleton(o: Object): Set  
+singletonList(o: Object): List  
+singletonMap(key: Object, value: Object): Map  
+unmodifiableCollection(c: Collection): Collection  
+unmodifiableList(list: List): List  
+unmodifiableMap(m: Map): Map  
+unmodifiableSet(s: Set): Set  
+unmodifiableSortedMap(s: SortedMap): SortedMap  
+unmodifiableSortedSet(s: SortedSet): SortedSet
```

Returns an immutable set containing the specified object.  
Returns an immutable list containing the specified object.  
Returns an immutable map with the key and value pair.  
Returns a read-only view of the collection.  
Returns a read-only view of the list.  
Returns a read-only view of the map.  
Returns a read-only view of the set.  
Returns a read-only view of the sorted map.  
Returns a read-only view of the sorted set.