Chapter 20 Collection Framework: Sets , Lists and Maps

Exercise 09 编程练习题(12.31 电邮给助教）

题一、

*24.3 （实现双向链表）程序清单 24-6 中使用的 MyLinkedList 类创建了一个单向链表，它只能单向遍历线性表。修改 Node 类，添加一个名为 previous 的数据域，让它指向链表中的前一个结点，如下所示：

```
public class Node<E> {
  E element;
  Node<E> next;
  Node<E> previous;

  public Node(E e) {
    element = e;
  }
}
```

实现一个名为 TwoWayLinkedList 的新类，使用双向链表来存储元素。课本中的 MyLinkedList 类继承自 MyAbstractList。定义 TwoWayLinkedList 继承 java.util.AbstractSequentialList 类。不光要实现 listIterator() 和 listIterator(int index) 方法，还要实现定义在 MyLinkedList 中包括的所有方法。都返回一个 java.util.ListIterator<E> 类型的实例。前者指向线性表的头部，后者指向指定下标的元素。

Exercise 09 编程练习题(12.31 电邮给助教）

题二、

*24.11 （动画：双向链表）编写一个程序，用动画实现双向链表的查找、插入和删除，如图 24-24 所示。按钮 Search 用来查找一个指定的值是否在线性表中；按钮 Delete 用来从线性表中删除一个特定值；按钮 Insert 用来在链表的特定下标处插入一个值，如果没有指定下标，则添加到链表的末尾。同时，添加两个名为 Forward Traversal 和 Backward Traversal 的按钮，用于采用遍历器分别以向前和往后的方式来显示元素。

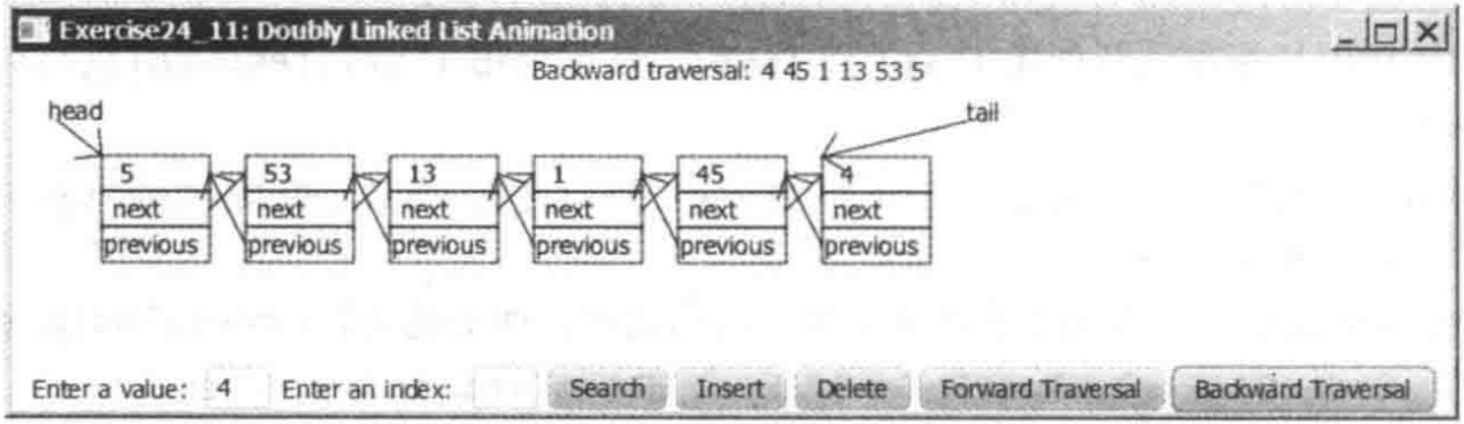


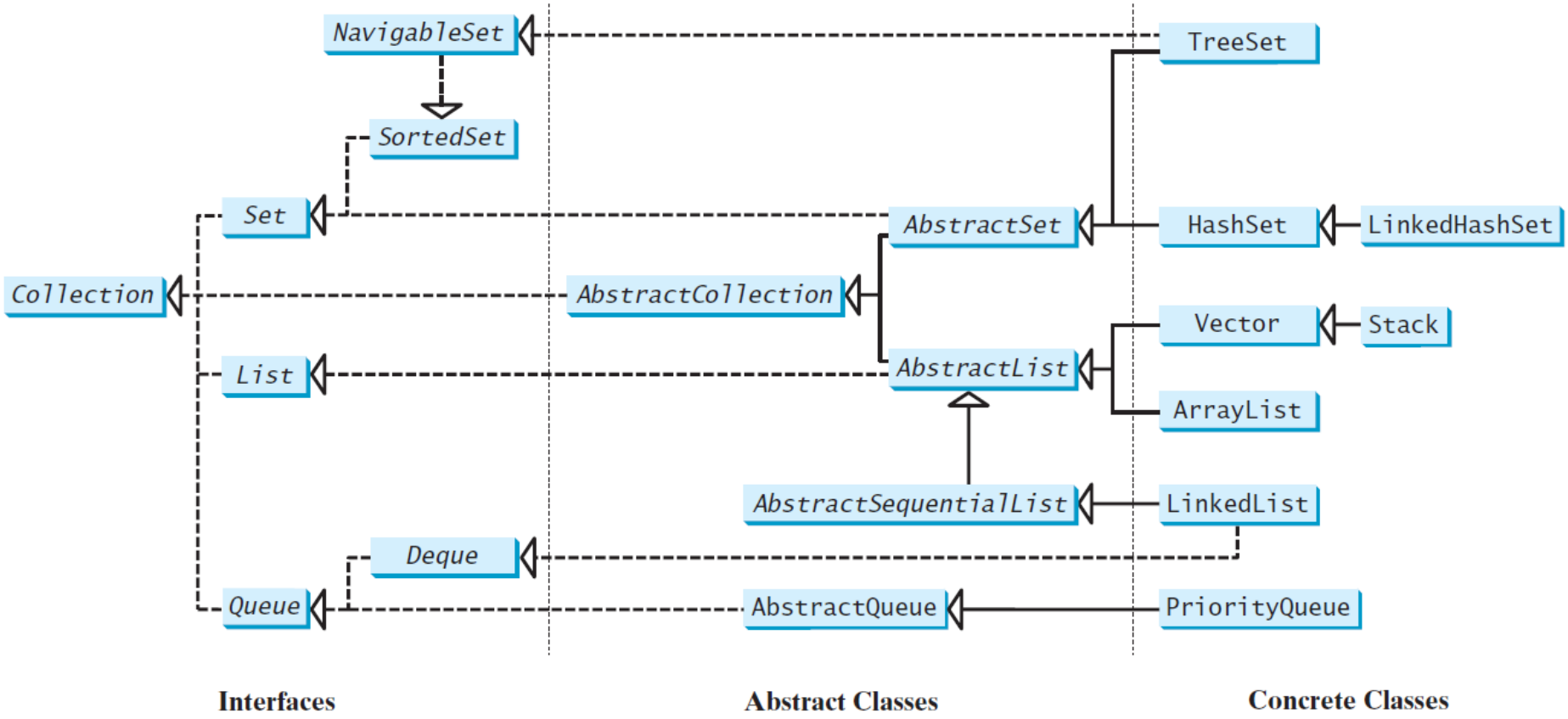
图 24-24　程序实现双向链表的运行动画

Java Collection Framework hierarchy

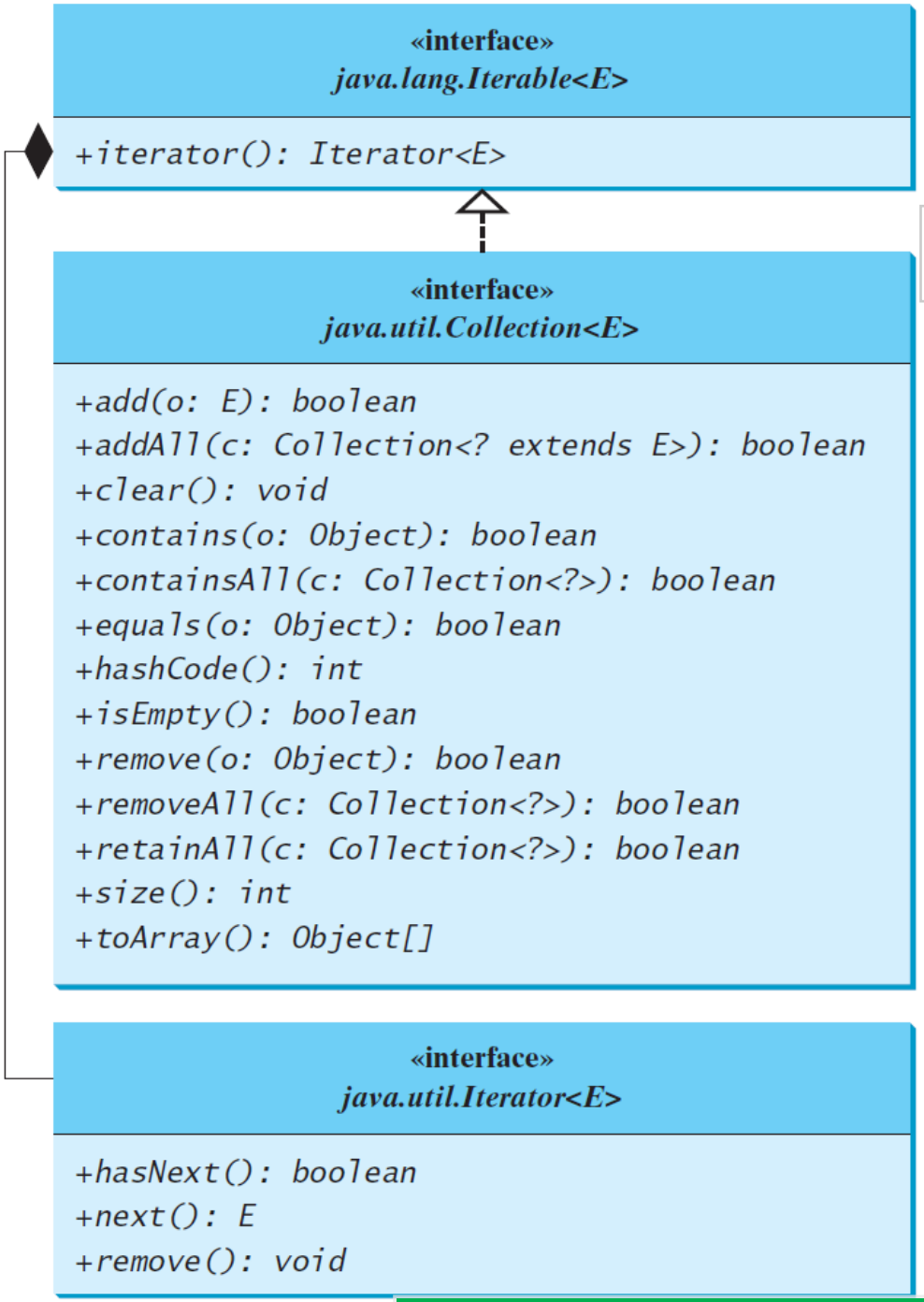
A *collection* is a container object that holds a group of objects, often referred to as *elements*. The Java Collections Framework supports three types of collections, named *lists, sets,* and *maps*.

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.

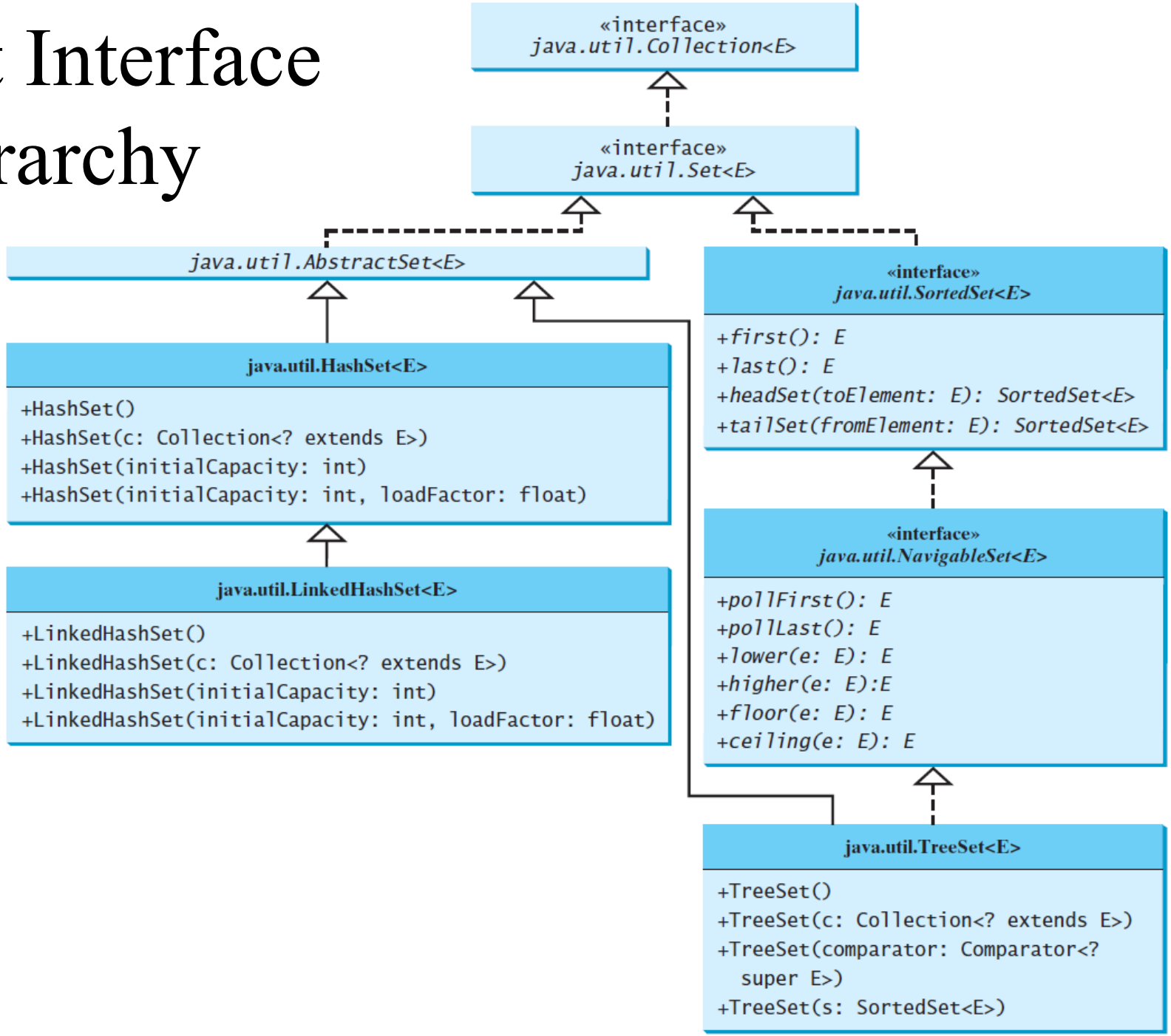
Returns true if this iterator has more elements to traverse.
Returns the next element from this iterator.
Removes the last element obtained using the next method.

TestMethodInCollection

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

TestSetWithEqualHashCode

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

Example: The Using Comparator to Sort Elements in a Set

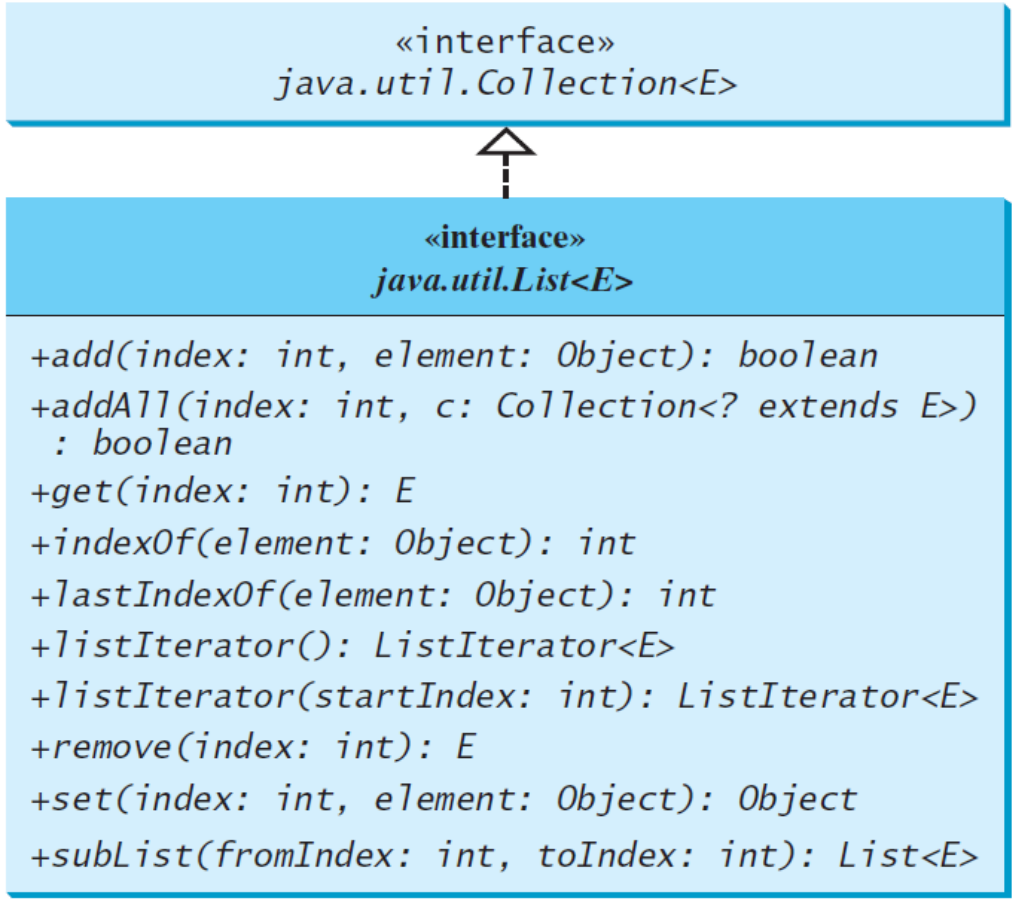
Write a program that demonstrates how to sort elements in a tree set using the Comparator interface. The example creates a tree set of geometric objects. The geometric objects are sorted using the compare method in the Comparator interface.

TestTreeSetWithComparator

The List Interface

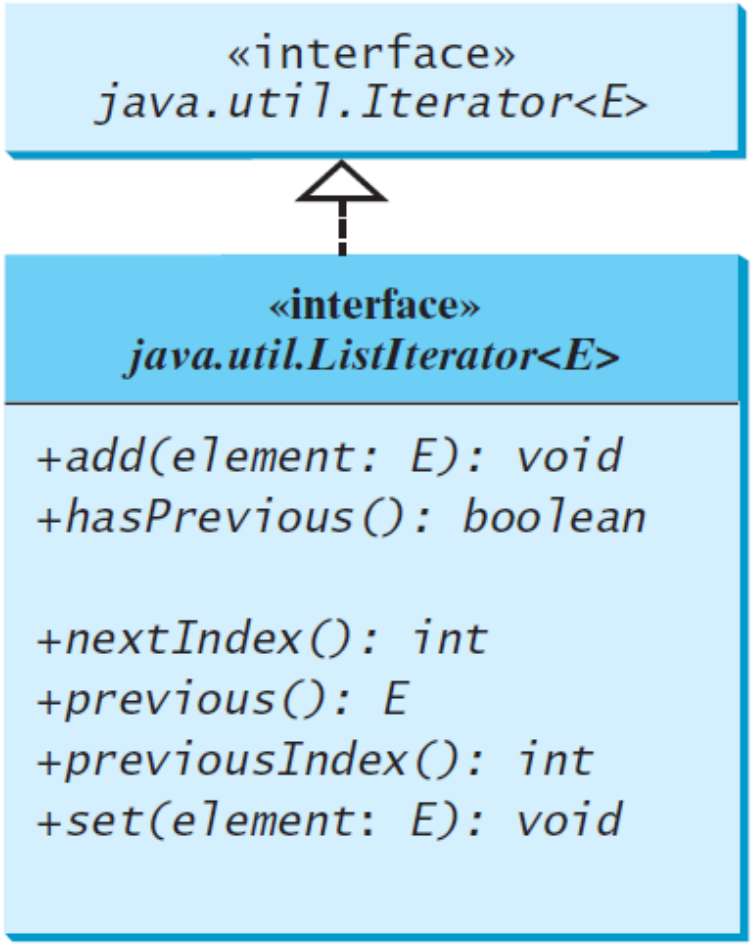
A list stores elements in a sequential order, and allows the user to specify where the element is stored. The user can access the elements by index.

The List Interface, cont.



`+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



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

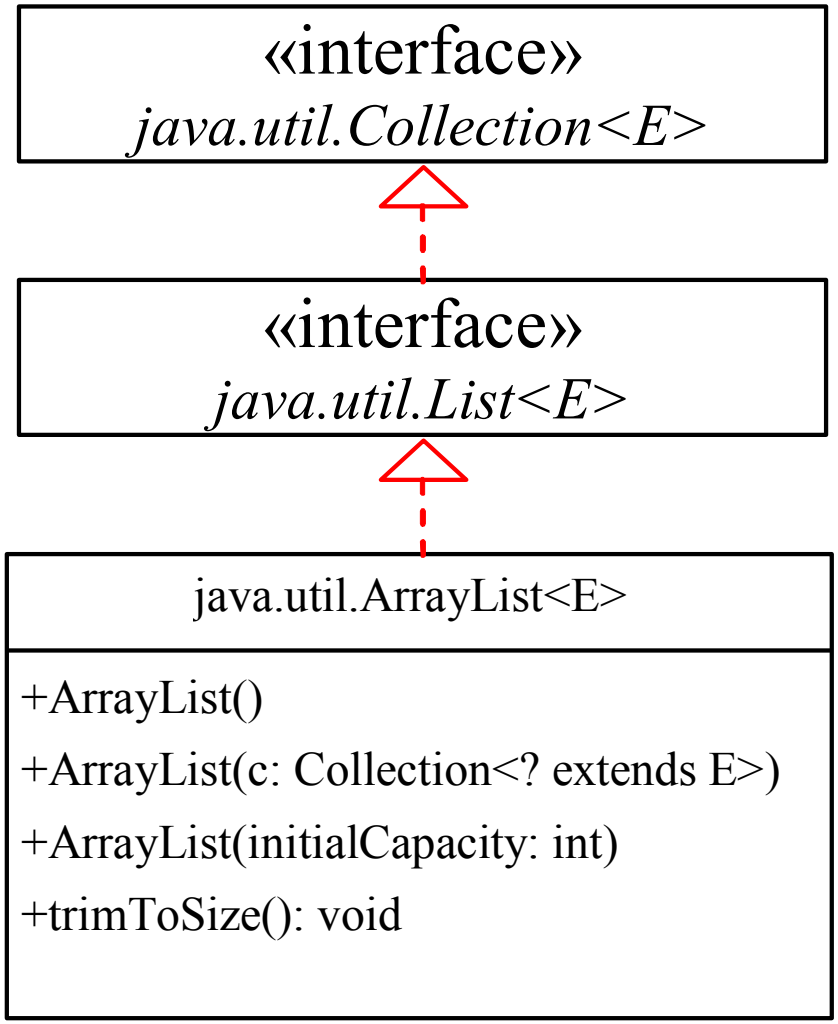
The ArrayList class and the LinkedList class are concrete implementations of the List interface.

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.

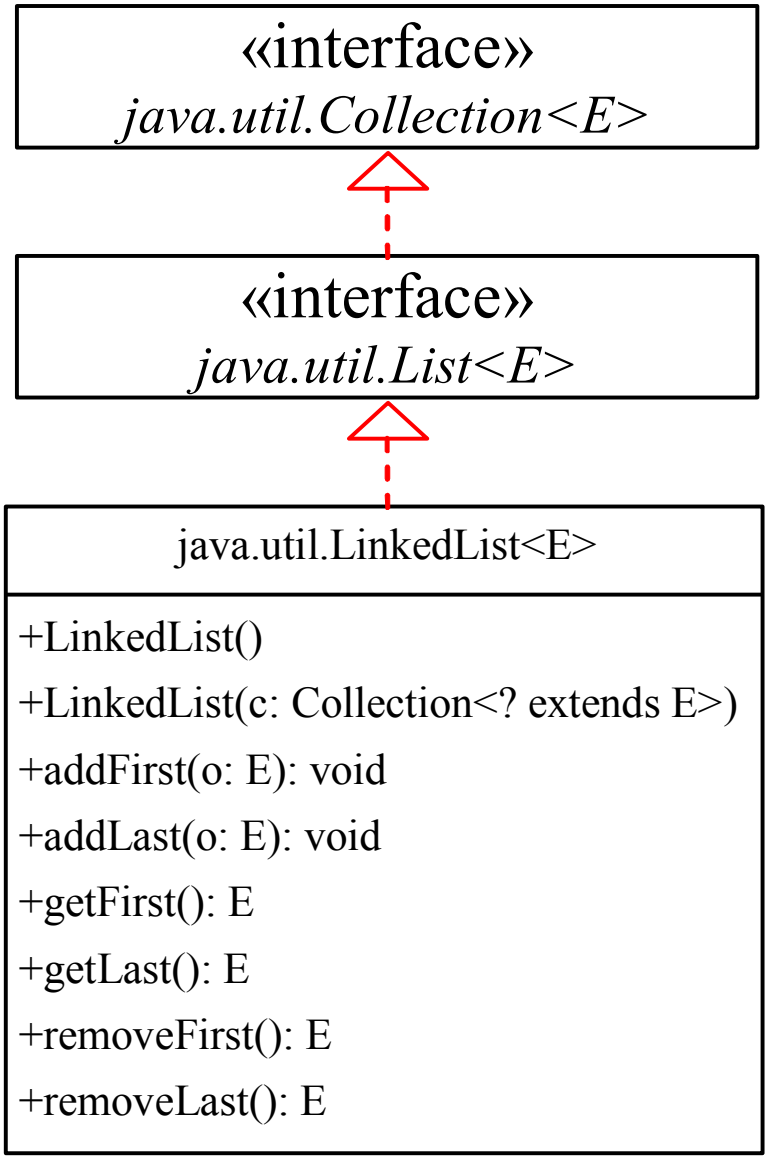
Creates an array list from an existing collection.

Creates an empty list with the specified initial capacity.

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

https://liveexample.pearsoncmg.com/dsanimation/ArrayListeBook.html

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.

https://liveexample.pearsoncmg.com/dsanimation/LinkedListeBook.html

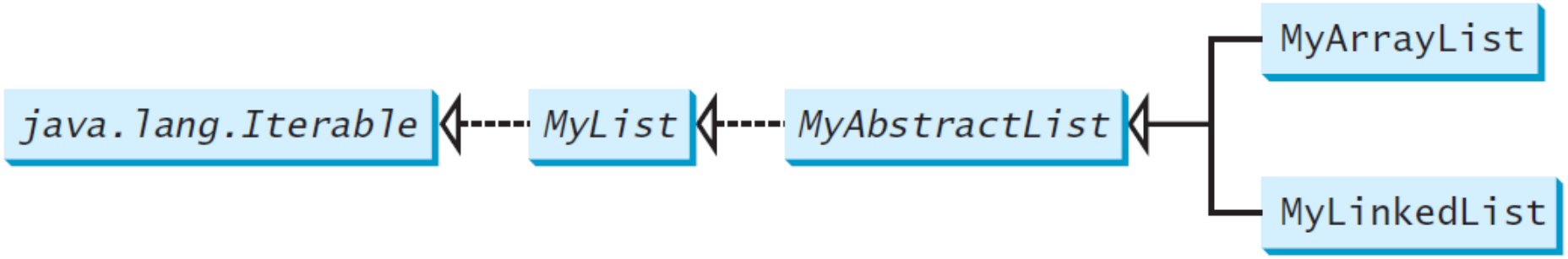
Example: Using ArrayList and LinkedList

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.

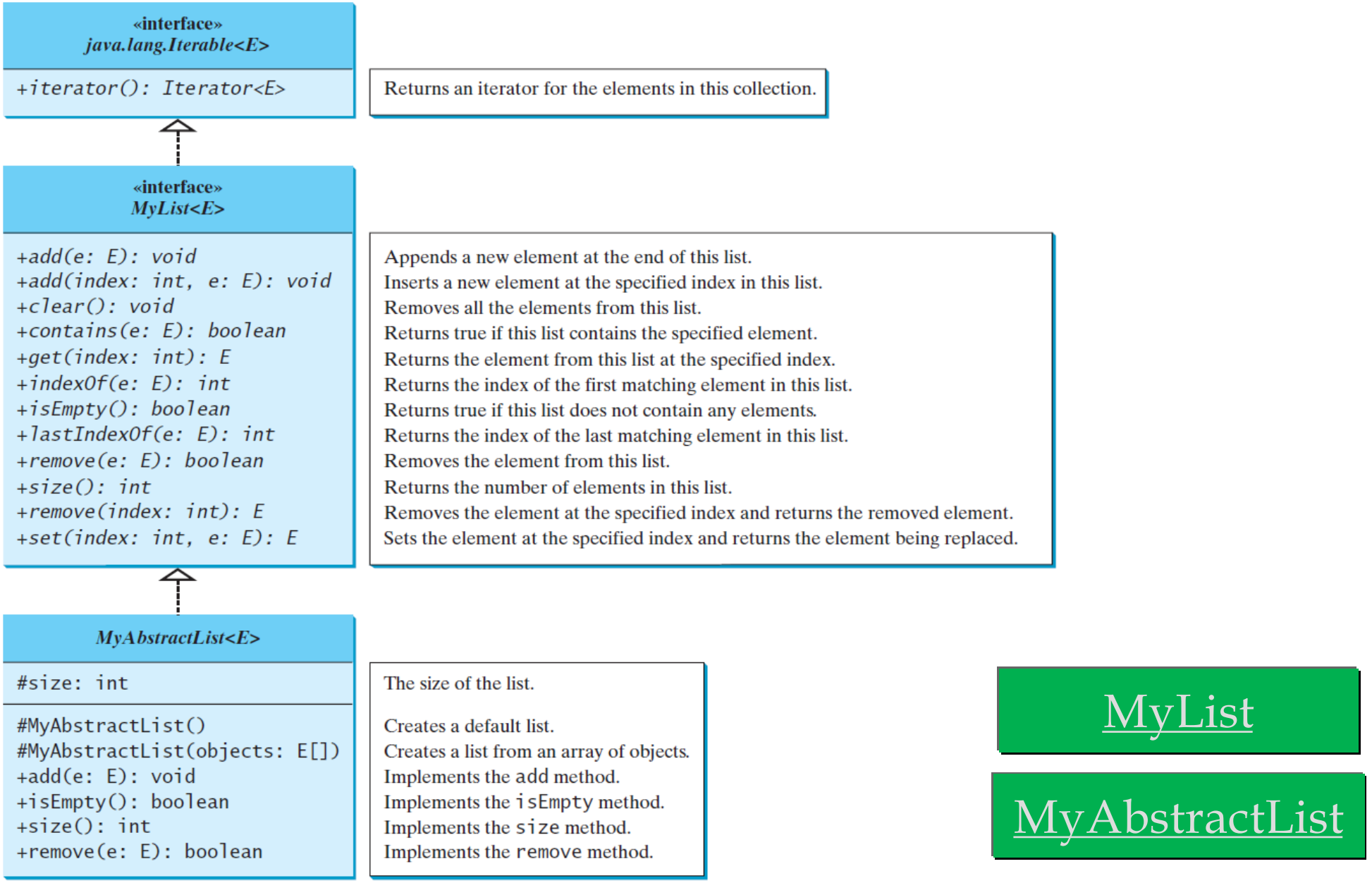
TestArrayAndLinkedList

Design of ArrayList and LinkedList

For convenience, let's name these two classes: MyArrayList and MyLinkedList. These two classes have common operations, but different data fields. The common operations can be generalized in an interface or an abstract class. A good strategy is to combine the virtues of interfaces and abstract classes by providing both interface and abstract class in the design so the user can use either the interface or the abstract class whichever is convenient. Such an abstract class is known as a convenience class.



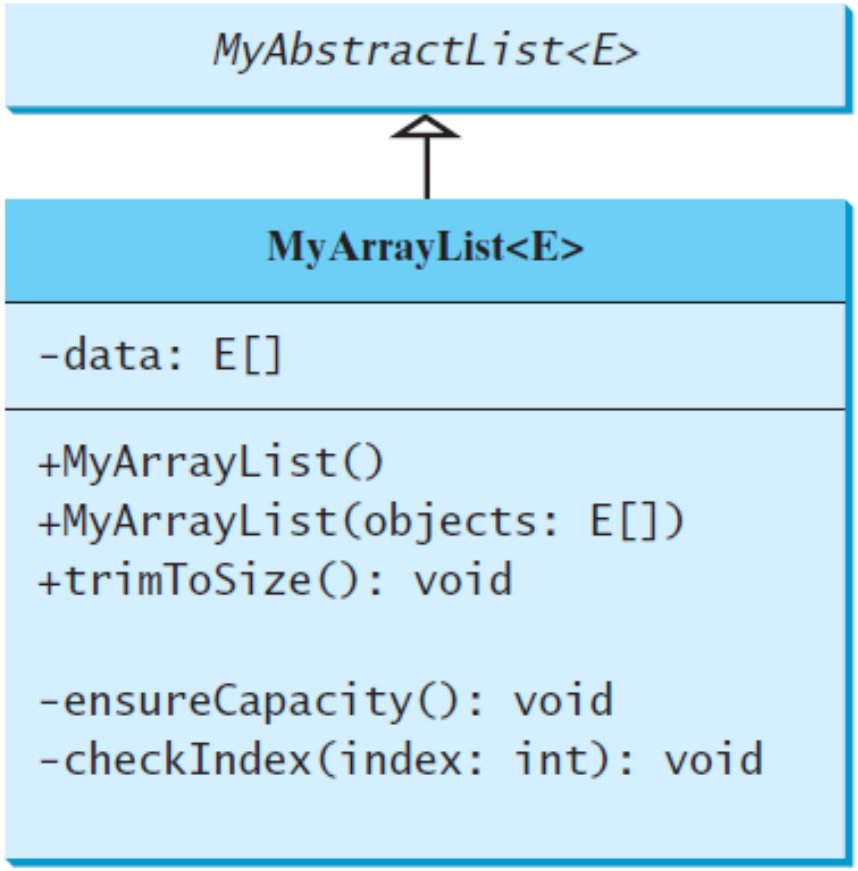
MyList Interface and MyAbstractList Class



MyList

MyAbstractList

Implementing MyArrayList



Creates a default array list.

Creates an array list from an array of objects.

Trims the capacity of this array list to the list's current size.

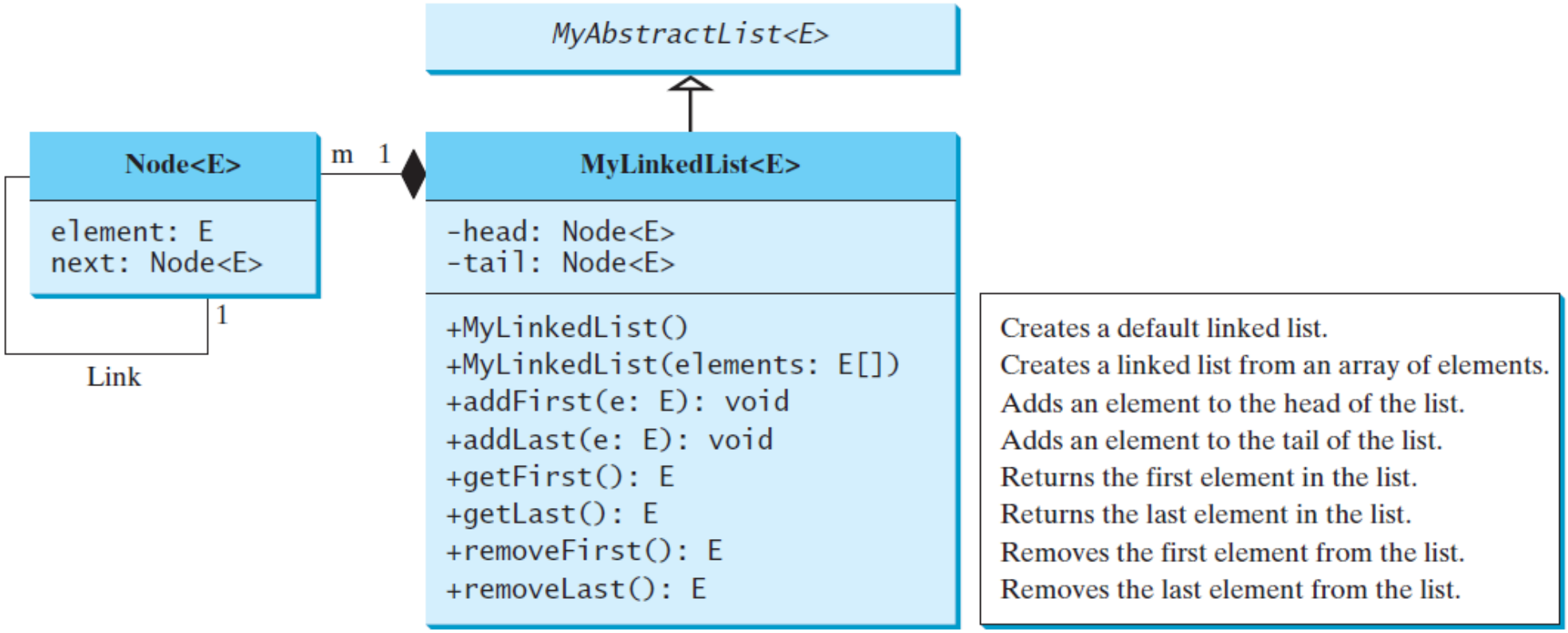
Doubles the current array size if needed.

Throws an exception if the index is out of bounds in the list.

MyArrayList

TestMyArrayList

MyLinkedList



MyLinkedList

TestMyLinkedList

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.

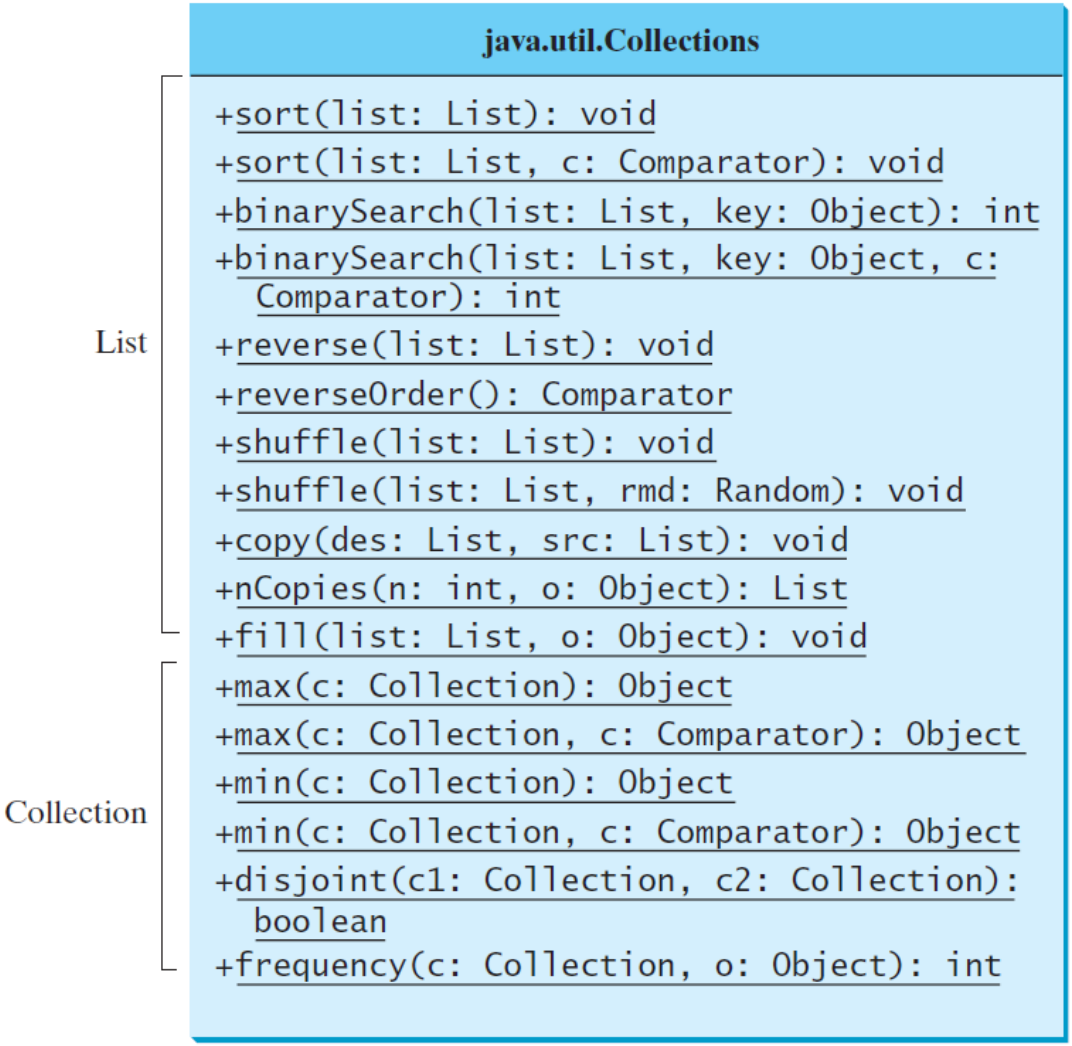
GeometricObjectComparator

TestComparator

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



Sorts the specified list.

Sorts the specified list with the comparator.

Searches the key in the sorted list using binary search.

Searches the key in the sorted list using binary search with the comparator.

Reverses the specified list.

Returns a comparator with the reverse ordering.

Shuffles the specified list randomly.

Shuffles the specified list with a random object.

Copies from the source list to the destination list.

Returns a list consisting of *n* copies of the object.

Fills the list with the object.

Returns the `max` object in the collection.

Returns the `max` object using the comparator.

Returns the `min` object in the collection.

Returns the `min` object using the comparator.

Returns true if `c1` and `c2` have no elements in common.

Returns the number of occurrences of the specified element in the collection.

Implementing Stacks and Queues

Using an array list to implement Stack

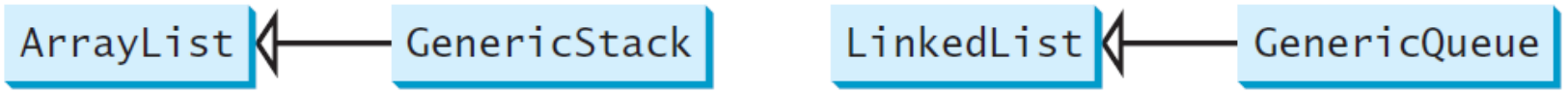
Use a linked list to implement Queue

Since the insertion and deletion operations on a stack are made only at the end of the stack, using an array list to implement a stack is more efficient than a linked list. Since deletions are made at the beginning of the list, it is more efficient to implement a queue using a linked list than an array list. This section implements a stack class using an array list and a queue using a linked list.

Design of the Stack and Queue Classes

There are two ways to design the stack and queue classes:

- Using inheritance: You can define the stack class by extending the array list class, and the queue class by extending the linked list class.



(a) Using inheritance

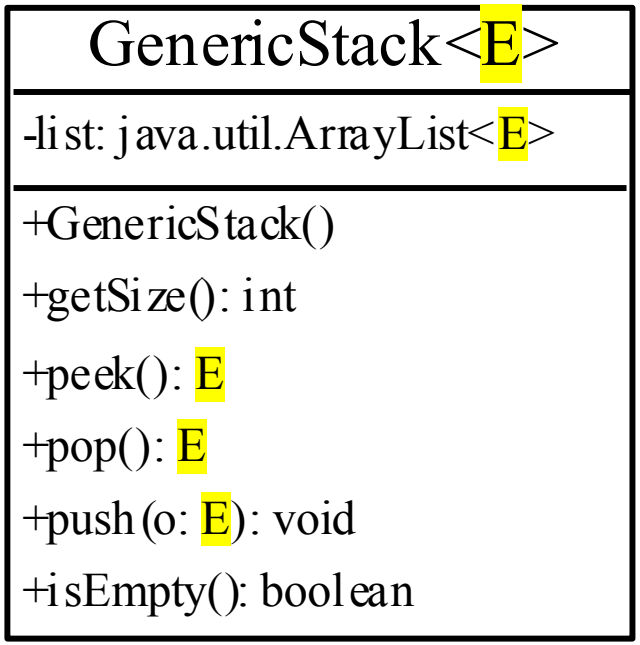
- Using composition: You can define an array list as a data field in the stack class, and a linked list as a data field in the queue class.

(b) Using composition

Composition is Better

Both designs are fine, but using composition is better because it enables you to define a complete new stack class and queue class without inheriting the unnecessary and inappropriate methods from the array list and linked list.

MyStack and MyQueue



GenericStack

An array list to store elements.

Creates an empty stack.

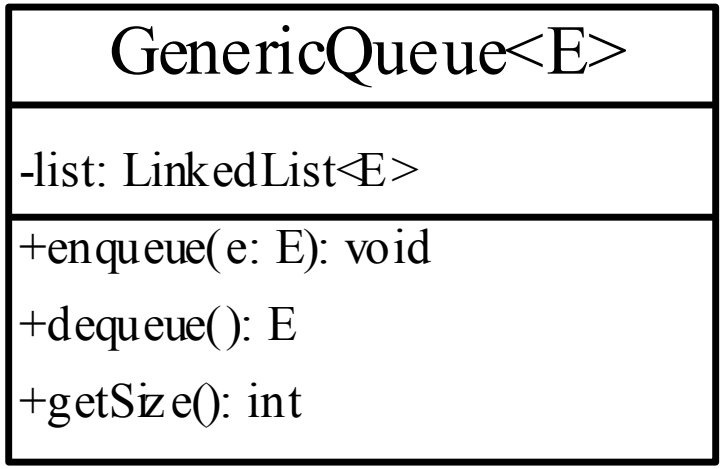
Returns the number of elements in this stack.

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 true if the stack is empty.



GenericQueue

Adds an element to this queue.

Removes an element from this queue.

Returns the number of elements from this queue.

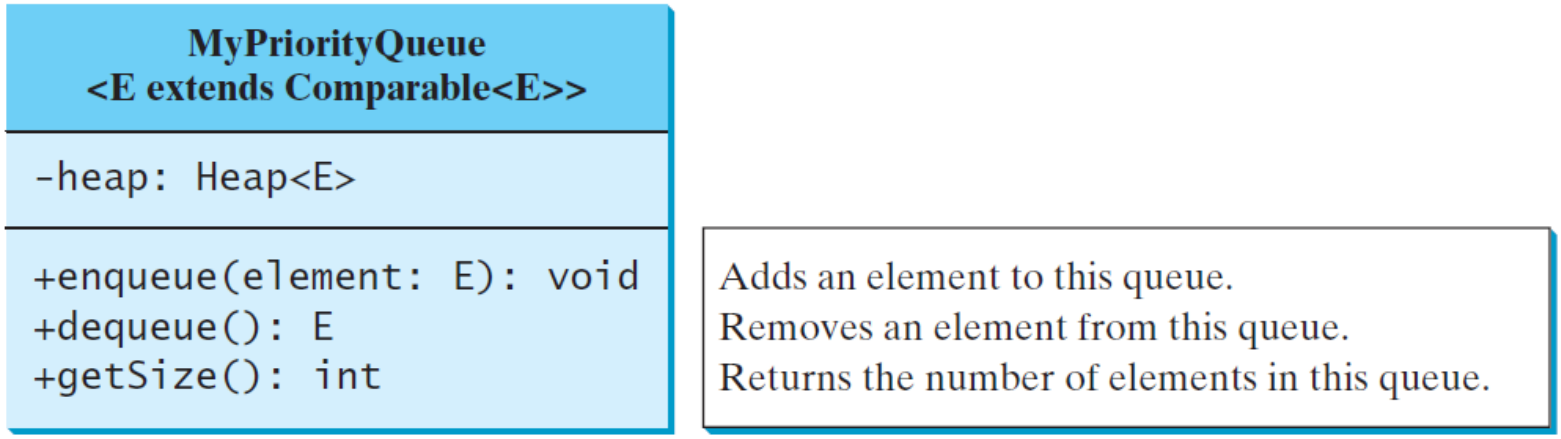
Example: Using Stacks and Queues

Write a program that creates a stack using MyStack and a queue using MyQueue. It then uses the push (enqueu) method to add strings to the stack (queue) and the pop (dequeue) method to remove strings from the stack (queue).

TestStackQueue

Priority Queue

A regular queue is a first-in and 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 with priorities. When accessing elements, the element with the highest priority is removed first. A priority queue has a largest-in, first-out behavior. For example, the emergency room in a hospital assigns patients with priority numbers; the patient with the highest priority is treated first.



MyPriorityQueue

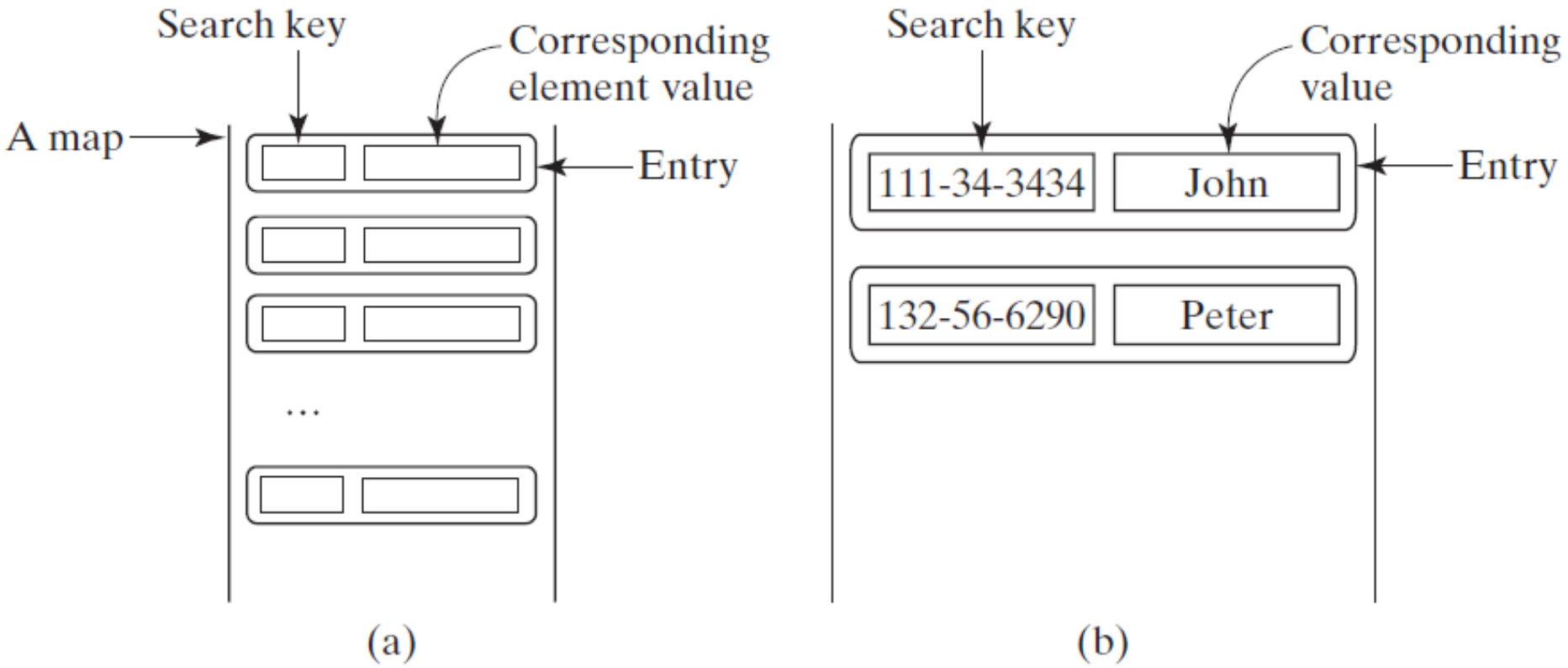
TestPriorityQueue

Performance of Sets and Lists

SetListPerformanceTest

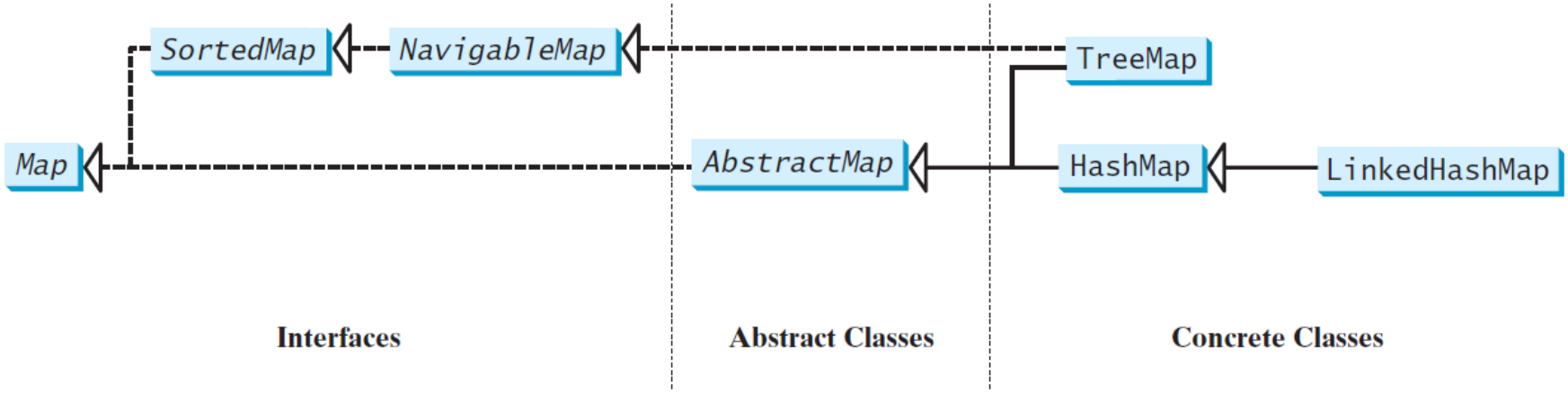
The Map Interface

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.

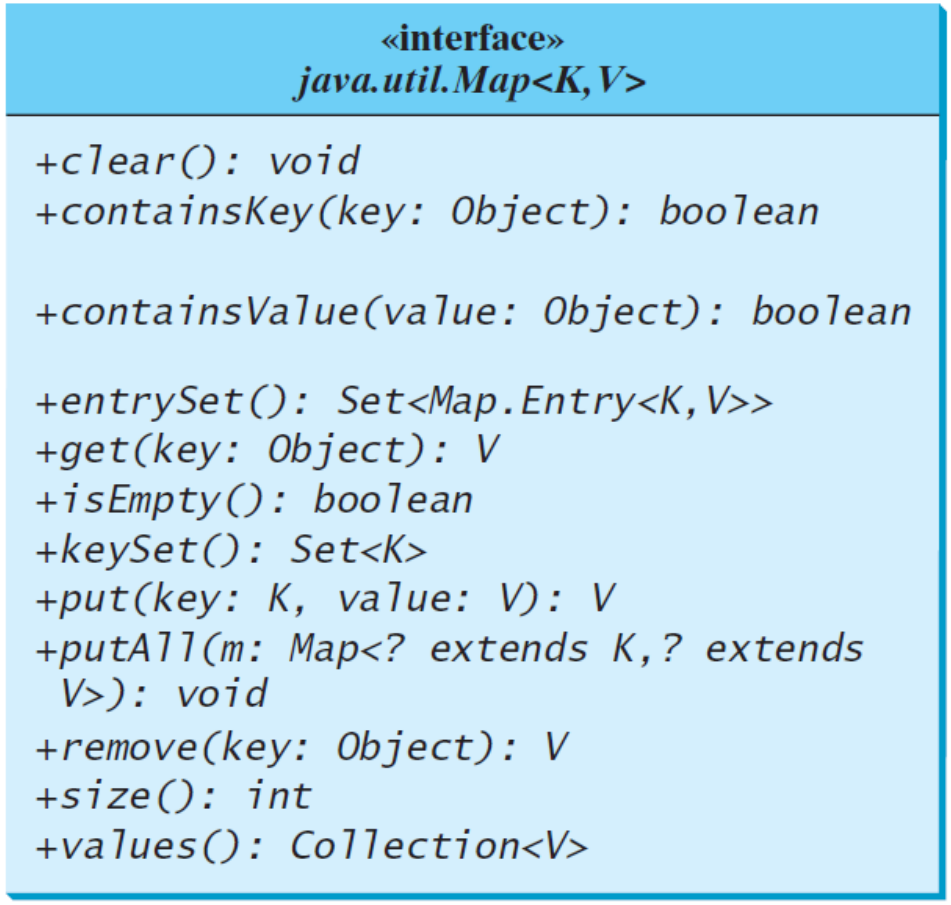


Map Interface and Class Hierarchy

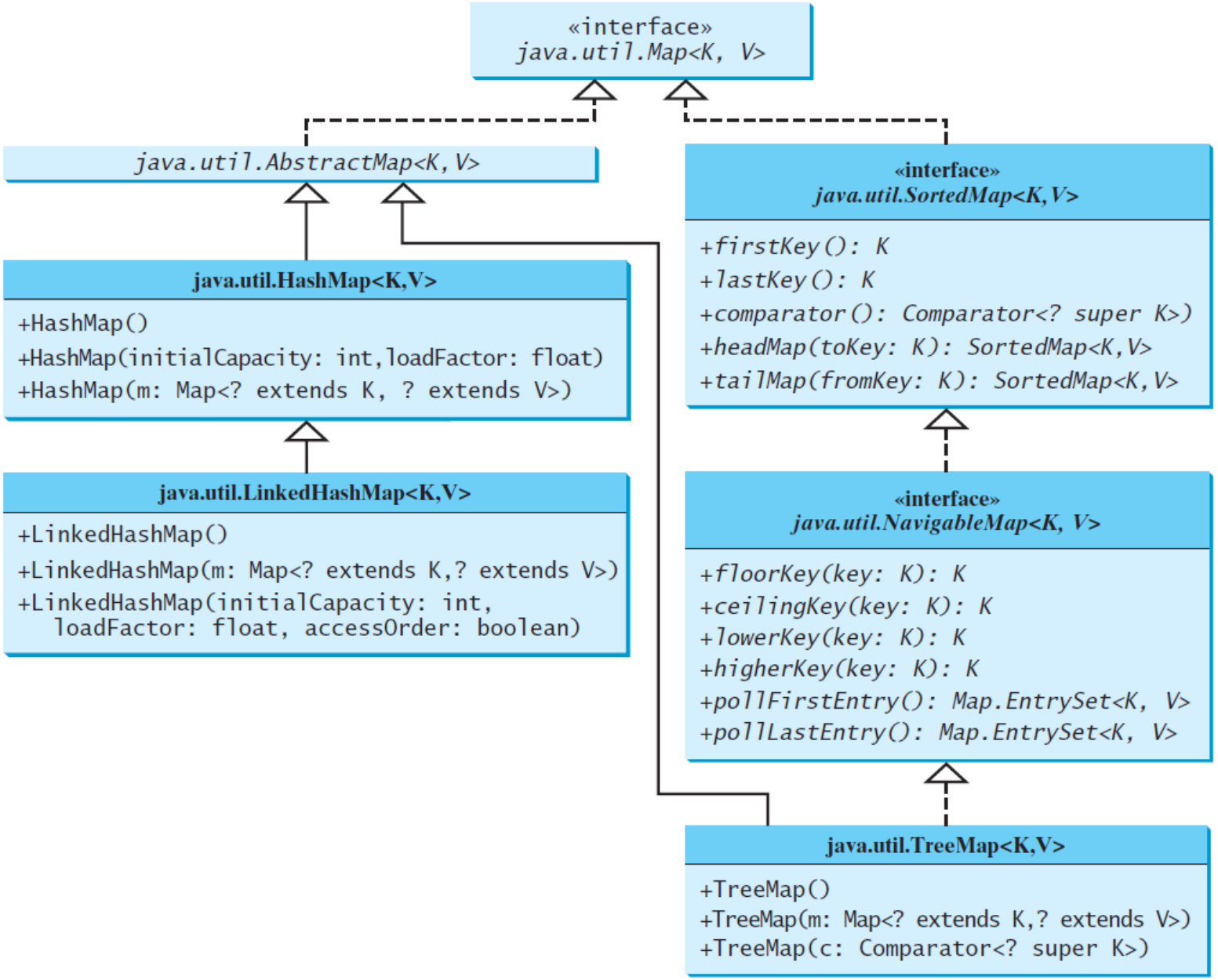
An instance of Map represents a group of objects, each of which is associated with a key. You can get the object from a map using a key, and you have to use a key to put the object into the map.



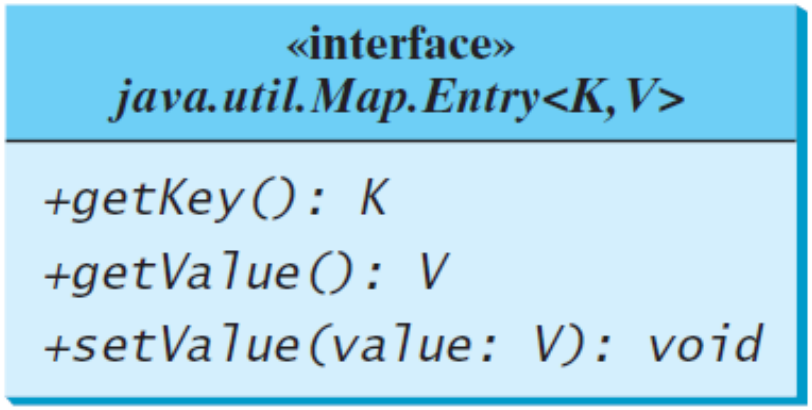
The Map Interface UML Diagram



Concrete Map Classes



Entry



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 Unmodiable Collections