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

If you create a `TreeSet` using its no-arg constructor, the **compareTo** meth the elements in the set, assuming that the class of the elements impleme interface. To use a comparator, you have to use the constructor **TreeSet parator)** to create a sorted set that uses the **compare** method in the comp ments in the set.

Listing 22.6 gives a program that demonstrates how to sort elements i **Comparator** interface. The example creates a tree set of geometric objec geometric objects are sorted using the **compare** method in the **Compara**

LISTING 22.6 TestTreeSetWithComparator.jav

tree set

display elements

```
1 import java.util.*;
2
3 public class TestTreeSetWithComparator {
4   public static void main(String[] args) {
5     // Create a tree set for geometric objects using a
6     Set<GeometricObject> set =
7       new TreeSet<GeometricObject>(new GeometricObject
8     set.add(new Rectangle(4, 5));
9     set.add(new Circle(40));
10    set.add(new Circle(40));
11    set.add(new Rectangle(4, 1));
12
13    // Display geometric objects in the tree set
14    System.out.println("A sorted set of geometric obje
15    for (GeometricObject element: set)
16      System.out.println("area = " + element.getArea()
17  }
18 }
```

```
A sorted set of geometric objects
area = 4.0
area = 20.0
area = 5022.548245743669
```

The **Circle** and **Rectangle** classes were defined in §14.2, "Abstract C subclasses of **GeometricObject**.

Two circles of the same radius are added to the set in the tree set (lines is stored, because the two circles are equal and the set does not allow du

Comparable vs. **Comparator**

Note

Comparable is used to compare the objects of the class that implemen **Comparator** can be used to compare the objects of the class that doesn't impleme

22.6 Lists

A set stores nonduplicate elements. To allow duplicate elements to be st you need to use a list. A list can not only store duplicate elements but al specify where they are stored. The user can access elements by an index. The `List` interface extends `Collection` to define an ordered collection with duplicates allowed. The `List` interface adds position-oriented operations, as well as a new list iterator that enables the user to traverse the list bidirectionally. The new methods in the `List` interface are shown in Figure 22.4.

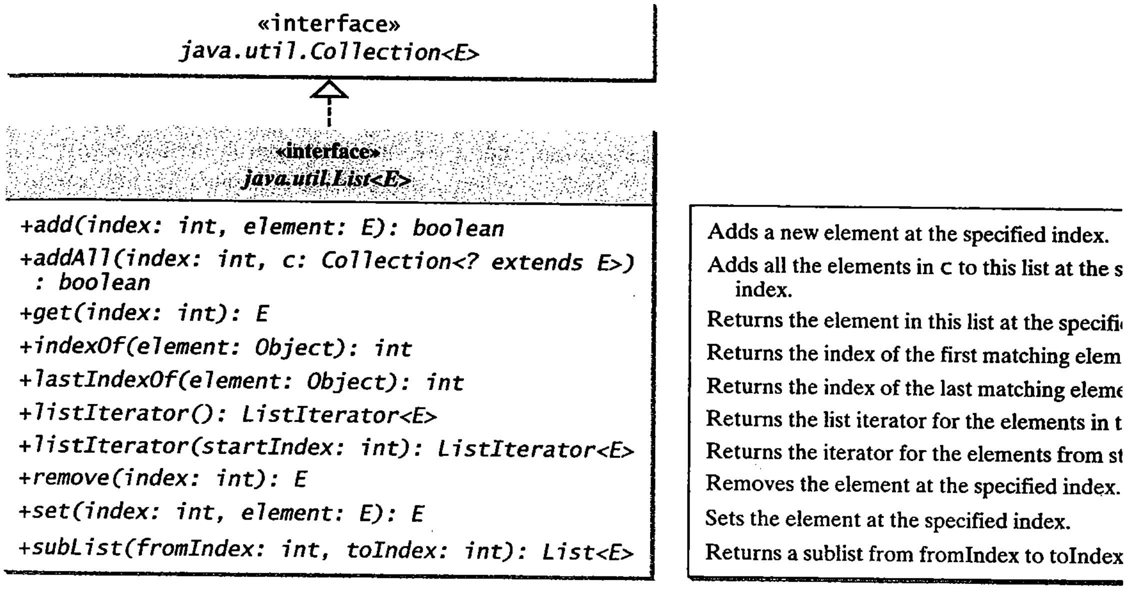

FIGURE 22.4 The `List` interface stores elements in sequence, permitting duplicates.

The `add(index, element)` method is used to insert an element at a specified index, and the `addAll(index, collection)` method to insert a collection at a specified index. The `remove(index)` method is used to remove an element at the specified index from the list. A new element can be set at the specified index using the `set(index, element)` method.

The `indexOf(element)` method is used to obtain the index of the specified element's first occurrence in the list, and the `lastIndexOf(element)` method to obtain the index of its last occurrence. A sublist can be obtained by using the `subList(fromIndex, toIndex)` method.

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. The methods in `ListIterator` are listed in Figure 22.5.

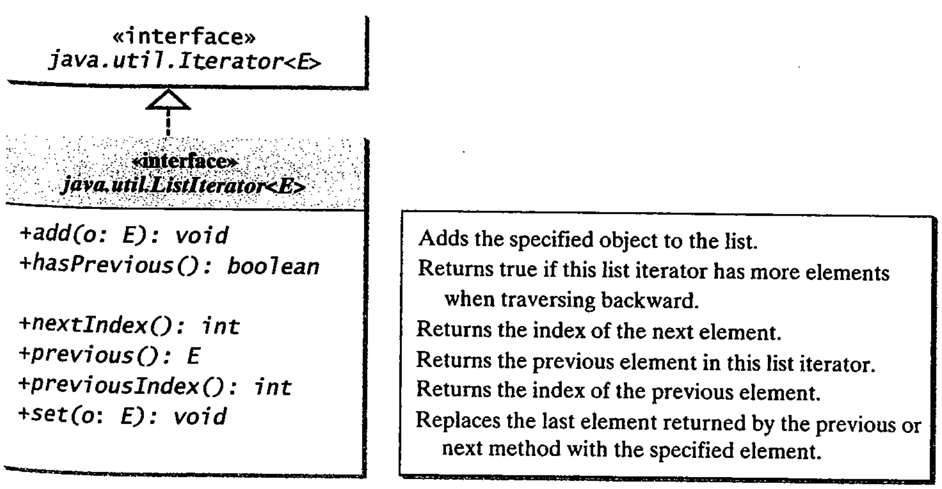

FIGURE 22.5 `ListIterator` enables traversal of a list bidirectionally.

The **add(element)** method inserts the specified element into tl inserted immediately before the next element that would be returned defined in the **Iterator** interface, if any, and after the element that w **previous()** method, if any. If the list contains no elements, the ne\ sole element on the list. The **set(element)** method can be used to ı returned by the **next** method or the **previous** method with the speci

The **hasNext()** method defined in the **Iterator** interface is use iterator has more elements when traversed in the forward direction, an method to check whether the iterator has more elements when trav direction.

The **next()** method defined in the **Iterator** interface returns t iterator, and the **previous()** method returns the previous eleme **nextIndex()** method returns the index of the next element in **previousIndex()** returns the index of the previous element in the it

The **AbstractList** class provides a partial implementation for t **AbstractSequentialList** class extends **AbstractList** to pro\ lists.

22.6.1 The ArrayList and LinkedList Classes

The **ArrayList** class (introduced in §11.11) and the **LinkedList** implementations of the **List** interface. **ArrayList** stores elements i dynamically created. If the capacity of the array is exceeded, a larger ı all the elements from the current array are copied to the new array. L ments in a linked list. Which of the two classes you use depends on yo need to support random access through an index without inserting except at the end, **ArrayList** offers the most efficient collection. If, tion requires the insertion or deletion of elements anywhere in the **LinkedList**. A list can grow or shrink dynamically. Once it is crea your application does not require the insertion or deletion of elemen efficient data structure.

trimToSize()

ArrayList is a resizable-array implementation of the **List** int methods for manipulating the size of the array used internally to stc Figure 22.6. Each **ArrayList** instance has a capacity, which is the s store the elements in the list. It is always at least as large as the lis added to an **ArrayList**, its capacity grows automatically. An **Arr** matically shrink. You can use the **trimToSize()** method to reduce t size of the list. An **ArrayList** can be constructed using i **ArrayList(Collection)**, or **ArrayList(initialCapacity)**.

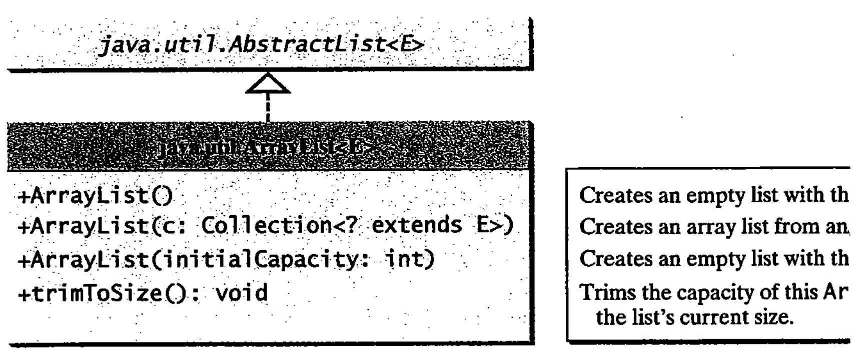

FIGURE 22.6 ArrayList implements List using an array.

`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, as shown in Figure 22.7. A `LinkedList` can be constructed using its no-arg constructor or `LinkedList(Collection)`.

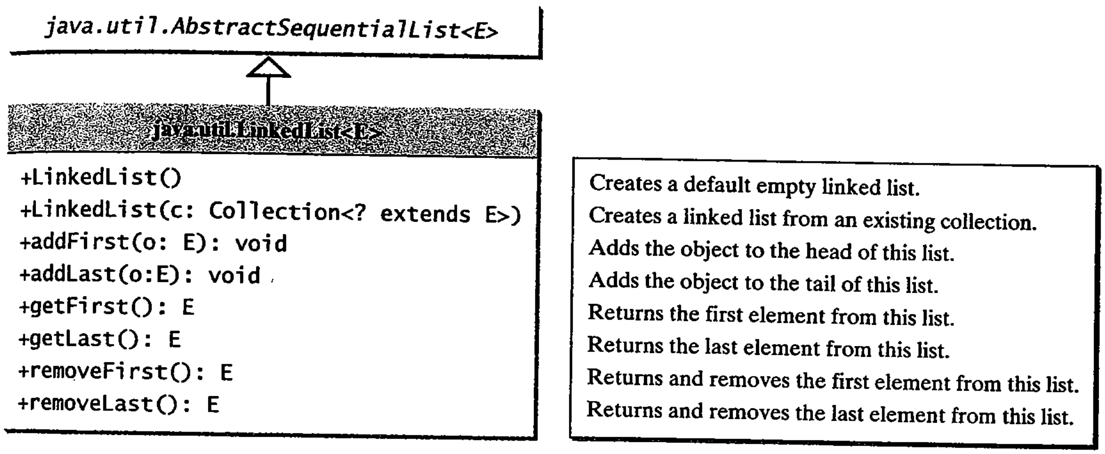

FIGURE 22.7 `LinkedList` provides methods for adding and inserting elements at both ends of the list.

Listing 22.7 gives a program that creates an array list filled with numbers and inserts new elements into specified locations in the list. The example also creates a linked list from the array list and inserts and removes elements from the list. Finally, the example traverses the list forward and backward.

LISTING 22.7 TestArrayAndLinkedList.java

```java
import java.util.*;

public class TestArrayAndLinkedList {
  public static void main(String[] args) {
    List<Integer> arrayList = new ArrayList<Integer>();
    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<Object>(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();
```

array list
linked list
list iterator

list iterator

```
    System.out.println("Display the linked list bacl
    listIterator = linkedList.listIterator(linkedLi:
    while (listIterator.hasPrevious()) {
      System.out.print(listIterator.previous() + " '
    }
  }
}
```

```
A list of integers in the array list:
[10, 1, 2, 30, 3, 1, 4]
Display the linked list forward:
green 10 red 1 2 30 3 1
Display the linked list backward:
1 3 30 2 1 red 10 green
```

A list can hold identical elements. Integer **1** is stored twice in the list (; and **LinkedList** are operated similarly. The critical difference bet internal implementation, which affects their performance. **ArrayList** ing elements and for inserting and removing elements from the end o: is efficient for inserting and removing elements anywhere in the list.

Tip

Arrays.asList(T... a) method

Java provides the static **asList** method for creating a list from a variable-ler a generic type. Thus you can use the following code to create a list of strings ;

```
List<String> list1 = Arrays.asList("red", "green",
List<Integer> list2 = Arrays.asList(10, 20, 30, 40
```

22.7 Static Methods for Lists and Collectic

You can use **TreeSet** to store sorted elements in a set. But there is n the Java Collections Framework provides static methods in the **Colle** be used to sort a list. The **Collections** class also contains the **bina** **shuffle**, **copy**, and **fill** methods on lists, and **max**, **min**, **disjo** methods on collections, as shown in Figure 22.8.

sort list

You can sort the comparable elements in a list in its natural order th method in the **Comparable** interface. You may also specify a compa For example, the following code sorts strings in a list.

```
List<String> list = Arrays.asList("red", "green", "bl
Collections.sort(list);
System.out.println(list);
```

The output is **[blue, green, red]**.

ascending order

descending order

The preceding code sorts a list in ascending order. To sort it in desc simply use the **Collections.reverseOrder()** method to return ; that orders the elements in reverse order. For example, the following co in descending order.

```
List<String> list = Arrays.asList("yellow", "red", "g
Collections.sort(list, Collections.reverseOrder());
System.out.println(list);
```

The output is **[yellow, red, green, blue]**.

	java.util.Collections	
List	+sort(list: List): void	Sorts the specified list.
	+sort(list: List, c: Comparator): void	Sorts the specified list with the comparat
	+binarySearch(list: List, key: Object): int	Searches the key in the sorted list using
	+binarySearch(list: List, key: Object, c: Comparator): int	Searches the key in the sorted list using with the comparator.
	+reverse(list: List): void	Reverses the specified list.
	+reverseOrder(): Comparator	Returns a comparator with the reverse o
	+shuffle(list: List): void	Shuffles the specified list randomly.
	+shuffle(list: List, rmd: Random): void	Shuffles the specified list with a random
	+copy(des: List, src: List): void	Copies from the source list to the destin
	+nCopies(n: int, o: Object): List	Returns a list consisting of *n* copies of th
	+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 compa
	+min(c: Collection): Object	Returns the `min` object in the collection
	+min(c: Collection, c: Comparator): Object	Returns the `min` object using the compa
	+disjoint(c1: Collection, c2: Collection): boolean	Returns true if `c1` and `c2` have no elem
	+frequency(c: Collection, o: Object): int	Returns the number of occurrences of th element in the collection.

FIGURE 22.8 The `Collections` class contains static methods for manipulating lists and collections.

You can use the `binarySearch` method to search for a key in a list. The list must be presorted in increasing order. If the key is not in the list, the method returns $-(insertion\ point + 1)$. Recall that the insertion point is where the item would fall in the list if it were present. For example, the following code searches the keys in a list of integers and a list of strings. **binarySe**

```
List<Integer> list1 =
  Arrays.asList(2, 4, 7, 10, 11, 45, 50, 59, 60, 66);
System.out.println("(1) Index: " + Collections.binarySearch(list1, 7));
System.out.println("(2) Index: " + Collections.binarySearch(list1, 9));

List<String> list2 = Arrays.asList("blue", "green", "red");
System.out.println("(3) Index: " +
  Collections.binarySearch(list2, "red"));
System.out.println("(4) Index: " +
  Collections.binarySearch(list2, "cyan"));
```

The output of the preceding code is

```
(1) Index: 2
(2) Index: -4
(3) Index: 2
(4) Index: -2
```

You can use the `reverse` method to reverse the elements in a list. For example, the following code displays `[blue, green, red, yellow]`. **reverse**

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

shuffle

You can use the **shuffle(List)** method to randomly reorder the el example, the following code shuffles the elements in **list**.

```
List<String> list = Arrays.asList("yellow", "red", "gre
Collections.shuffle(list);
System.out.println(list);
```

You can also use the **shuffle(List, Random)** method to randomly rec a list with a specified **Random** object. Using a specified **Random** object is list with identical sequences of elements for the same original list. For ex code shuffles the elements in **list**.

```
List<String> list1 = Arrays.asList("yellow", "red", "gr
List<String> list2 = Arrays.asList("yellow", "red", "gr
Collections.shuffle(list1, new Random(20));
Collections.shuffle(list2, new Random(20));
System.out.println(list1);
System.out.println(list2);
```

You will see that **list1** and **list2** have the same sequence of elements shuffling.

copy

You can use the **copy(det, src)** method to copy all the elements fr destination list on the same index. The destination must be as long as th longer, the remaining elements in the source list are not affected. For exa code copies **list2** to **list1**.

```
List<String> list1 = Arrays.asList("yellow", "red", "gr
List<String> list2 = Arrays.asList("white", "black");
Collections.copy(list1, list2);
System.out.println(list1);
```

The output for **list1** is **[white, black, green, blue]**. The **copy** shallow copy. Only the references of the elements from the source list are

nCopies

You can use the **nCopies(int n, Object o)** method to create an imm sists of **n** copies of the specified object. For example, the following code five **Calendar** objects.

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

The list created from the **nCopies** method is immutable, so you cann update elements in the list. All the elements have the same references.

fill

You can use the **fill(List list, Object o)** method to replace all the with the specified element. For example, the following code displays **[black**

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

max and min methods

You can use the **max** and **min** methods for finding the maximum and mini collection. The elements must be comparable using the **Comparable** **Comparator** interface. For example, the following code displays the la strings in a collection.

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

The **disjoint(collection1, collection2)** method returns **true** if the two collections have no elements in common. For example, in the following code, **disjoint(collection1, collection2)** returns **false**, but **disjoint(collection1, collection3)** returns **true**.

disjoint m

```
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));
System.out.println(Collections.disjoint(collection1, collection3));
```

The **frequency(collection, element)** method finds the number of occurrences of the element in the collection. For example, **frequency(collection, "red")** returns 2 in the following code.

frequency

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

22.8 Performance of Sets and Lists

We now conduct an interesting experiment to test the performance of sets and lists. Listing 22.8 gives a program that shows the execution time of adding and removing elements in a hash set, linked hash set, tree set, array list, and linked list.

LISTING 22.8 SetListPerformanceTest.java

```
import java.util.*;

public class SetListPerformanceTest {
  public static void main(String[] args) {
    // Create a hash set, and test its performance
    Collection<Integer> set1 = new HashSet<Integer>();
    System.out.println("Time for hash set is " +
      getTestTime(set1, 500000) + " milliseconds");

    // Create a linked hash set, and test its performance
    Collection<Integer> set2 = new LinkedHashSet<Integer>();
    System.out.println("Time for linked hash set is " +
      getTestTime(set2, 500000) + " milliseconds");

    // Create a tree set, and test its performance
    Collection<Integer> set3 = new TreeSet<Integer>();
    System.out.println("Time for tree set is " +
      getTestTime(set3, 500000) + " milliseconds");

    // Create an array list, and test its performance
    Collection<Integer> list1 = new ArrayList<Integer>();
    System.out.println("Time for array list is " +
      getTestTime(list1, 60000) + " milliseconds");

    // Create a linked list, and test its performance
    Collection<Integer> list2 = new LinkedList<Integer>();
    System.out.println("Time for linked list is " +
      getTestTime(list2, 60000) + " milliseconds");
  }

  public static long getTestTime(Collection<Integer> c, int size) {
    long startTime = System.currentTimeMillis();

```

a hash set

a linked has

a tree set

an array list

a linked list

start time

```
34      // Add numbers 0, 1, 2, ..., size - 1 to the array
35      List<Integer> list = new ArrayList<Integer>();
36      for (int i = 0; i < size; i++)
37        list.add(i);
38
39      Collections.shuffle(list); // Shuffle the array l·
40
41      // Add the elements to the container
42      for (int element: list)
43        c.add(element);
44
45      Collections.shuffle(list); // Shuffle the array l·
46
47      // Remove the element from the container
48      for (int element: list)
49        c.remove(element);
50
51      long endTime = System.currentTimeMillis();
52      return endTime - startTime; // Return the executi(
53    }
54 }
```

shuffle (line 39)
add to container (line 43)
shuffle (line 45)
remove from container (line 49)
end time (line 51)
return elapsed time (line 52)

```
Time for hash set is 1437 milliseconds
Time for linked hash set is 1891 milliseconds
Time for tree set is 2891 milliseconds
Time for array list is 13797 milliseconds
Time for linked list is 15344 milliseconds
```

The `getTestTime` method creates a list of distinct integers from 0 to `si`: shuffles the list (line 39), adds the elements from the list to a container c fles the list again (line 45), removes the elements from the container (line returns the execution time (line 52).

The program creates a hash set (line 6), a linked hash set (line 11), a array list (line 21), and a linked list (line 26). The program obtains th adding and removing 500000 elements in the three sets and adding and r ments in the two lists.

sets are better

As you see, sets are much more efficient than lists. If sets are sufficient use sets. Furthermore, if no particular order is needed for your applicati(

The program tested general remove operations for array lists and linl plexity is about the same. Please note that linked lists are more efficien insertion and deletion anywhere in the list except at the end.

22.9 The Vector and Stack Classes

The Java Collections Framework was introduced with Java 2. Several supported earlier, among them the `Vector` and `Stack` classes. These cla: to fit into the Java Collections Framework, but all their old-style meth compatibility.

`Vector` is the same as `ArrayList`, except that it contains synchronizec ing and modifying the vector. Synchronized methods can prevent data corr is accessed and modified by two or more threads concurrently. For the man not require synchronization, using `ArrayList` is more efficient than using

The `Vector` class implements the `List` interface. It also has the meth original `Vector` class defined prior to Java 2, as shown in Figure 22.9.

22.11 Maps

Suppose your program stores a million students and frequently searches for a student using the social security number. An efficient data structure for this task is the *map*. A map is a container that stores the elements along with the keys. The keys are like indexes. In `List`, the indexes are integers. In `Map`, the keys can be any objects. A map cannot contain duplicate keys. Each key maps to one value. A key and its corresponding value form an entry, which is actually stored in a map, as shown in Figure 22.14. why map?

There are three types of maps: **HashMap**, **LinkedHashMap**, and **TreeMap**. The common features of these maps are defined in the **Map** interface. Their relationship is shown in Figure 22.15.

The `Map` interface provides the methods for querying, updating, and obtaining a collection of values and a set of keys, as shown in Figure 22.16.

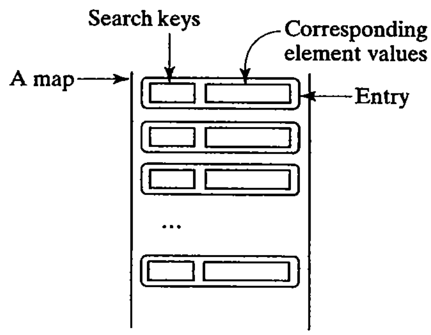

FIGURE 22.14 The entries consisting of key/value pairs are stored in a map.

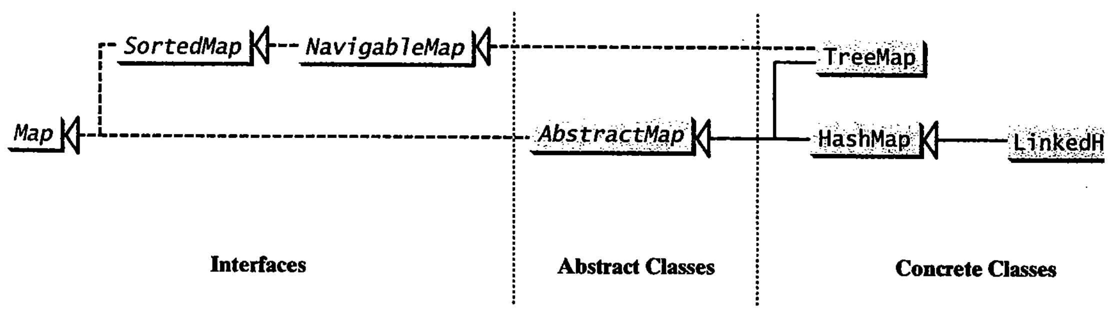

FIGURE 22.15 A map stores key/value pairs.

«interface» java.util.Map<K,V>	
+clear(): void	Removes all entries from this map.
+containsKey(key: Object): boolean	Returns true if this map contains entries for the specified key.
+containsValue(value: Object): boolean	Returns true if this map maps one or more keys to the specified value.
+entrySet(): Set<Map.Entry<K,V>>	Returns a set consisting of the entries in this map.
+get(key: Object): V	Returns the value for the specified key in this map.
+isEmpty(): boolean	Returns true if this map contains no entries.
+keySet(): Set<K>	Returns a set consisting of the keys in this map.
+put(key: K, value: V): V	Puts a mapping in this map.
+putAll(m: Map<? extends K,? extends V>): void	Adds all the entries from m to this map.
+remove(key: Object): V	Removes the entries for the specified key.
+size(): int	Returns the number of entries in this map.
+values(): Collection<V>	Returns a collection consisting of the values in this map.

FIGURE 22.16 The `Map` interface maps keys to values.

update methods

The *update methods* include clear, put, putAll, and remove. Th removes all entries from the map. The put(K key, V value) method as: a key in the map. If the map formerly contained a mapping for this key, ı ated with the key is returned. The putAll(Map m) method adds the sı map. The remove(Object key) method removes the map elements f from the map.

query methods

The *query methods* include containsKey, containsValue, isEm| containsKey(Object key) method checks whether the map contain specified key. The containsValue(Object value) method checks w tains a mapping for this value. The isEmpty() method checks whether tl mappings. The size() method returns the number of mappings in the m

keySet()
values()
entrySet()

You can obtain a set of the keys in the map using the keySet() meth of the values in the map using the values() method. The entrySet() of objects that implement the Map.Entry<K, V> interface, where Entry for the Map interface, as shown in Figure 22.17. Each object in the set is pair in the underlying map.

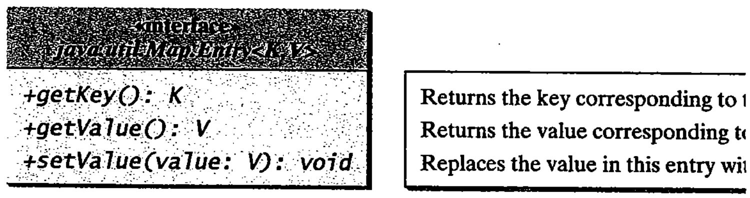

Figure 22.17 The Map.Entry interface operates on an entry in the map

AbstractMap

The AbstractMap class is a convenience class that implements all the interface except the entrySet() method.

The SortedMap interface extends the Map interface to maintain the m order of keys with additional methods firstKey() and lastKey() for and highest key, headMap(toKey) for returning the portion of the map than toKey, and tailMap(fromKey) for returning the portion of the n greater than or equal to fromKey.

concrete implementation

The HashMap, LinkedHashMap, and TreeMap classes are three *co tions* of the Map interface, as shown in Figure 22.18.

HashMap

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

LinkedHashMap

LinkedHashMap extends HashMap with a linked-list implementatic ordering of the entries in the map. The entries in a HashMap are not ordere a LinkedHashMap can be retrieved either in the order in which they we map (known as the *insertion order*) or in the order in which they were least recently to most recently accessed (*access order*). The no-arg cons LinkedHashMap with the insertion order. To construct a LinkedHashM order, use the LinkedHashMap(initialCapacity, loadFactor, tr

insertion order
access order

TreeMap

The TreeMap class is efficient for traversing the keys in a sorted ord sorted using the Comparable interface or the Comparator interface. If yo using its no-arg constructor, the compareTo method in the Comparable compare the elements in the map, assuming that the class of the eleme Comparable interface. To use a comparator, you have to use the Tree comparator) constructor to create a sorted map that uses the compare ı parator to order the elements in the map based on the keys.

SortedMap

SortedMap is a subinterface of Map, which guarantees that the entı sorted. Additionally, it provides the methods firstKey() and lastK the first and last keys in the map, and headMap(toKey) and tailM

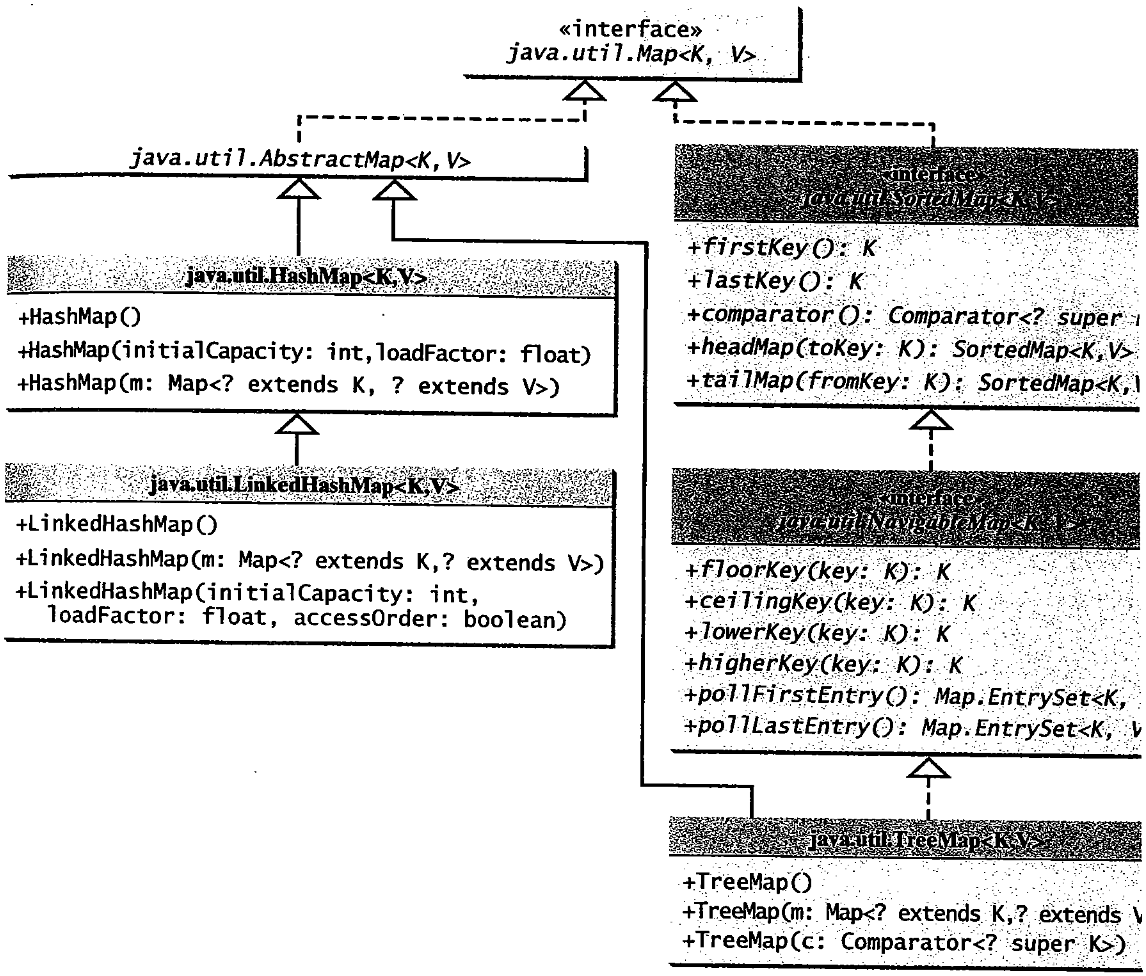

FIGURE 22.18 The Java Collections Framework provides three concrete map classes.

returning a portion of the map whose keys are less than `toKey` and greater than or equal to `fromKey`.

`NavigableMap` extends `SortedMap` to provide navigation methods `lowerKey(key)`, `floorKey(key)`, `ceilingKey(key)`, and `higherKey(key)` that return keys respectively less than, less than or equal, greater than or equal, and greater than a given key and return `null` if there is no such key. The `pollFirstEntry()` and `pollLastEntry()` methods remove and return the first and last entry in the tree map, respectively. **Navigable**

Note

Prior to Java 2, `java.util.Hashtable` was used for mapping keys with elements. `Hashtable` was redesigned to fit into the Java Collections Framework with all its methods retained for compatibility. `Hashtable` implements the `Map` interface and is used in the same way as `HashMap`, except that `Hashtable` is synchronized. **Hashtable**

Listing 22.11 gives an example that creates a hash map, a linked hash map, and a tree map that map students to ages. The program first creates a hash map with the student's name as its key and the age as its value. The program then creates a tree map from the hash map and displays the mappings in ascending order of the keys. Finally, the program creates a linked hash map, adds the same entries to the map, and displays the entries.

LISTING 22.11 TestMap.java

create map
add entry
tree map
linked hash map

```
import java.util.*;

public class TestMap {
  public static void main(String[] args) {
    // Create a HashMap
    Map<String, Integer> hashMap = new HashMap<String
    hashMap.put("Smith", 30);
    hashMap.put("Anderson", 31);
    hashMap.put("Lewis", 29);
    hashMap.put("Cook", 29);

    System.out.println("Display entries in HashMap");
    System.out.println(hashMap + "\n");

    // Create a TreeMap from the previous HashMap
    Map<String, Integer> treeMap =
      new TreeMap<String, Integer>(hashMap);
    System.out.println("Display entries in ascending (
    System.out.println(treeMap);

    // Create a LinkedHashMap
    Map<String, Integer> linkedHashMap =
      new LinkedHashMap<String, Integer>(16, 0.75f, tı
    linkedHashMap.put("Smith", 30);
    linkedHashMap.put("Anderson", 31);
    linkedHashMap.put("Lewis", 29);
    linkedHashMap.put("Cook", 29);

    // Display the age for Lewis
    System.out.println("The age for " + "Lewis is " +
      linkedHashMap.get("Lewis").intValue());

    System.out.println("\nDisplay entries in LinkedHas
    System.out.println(linkedHashMap);
  }
}
```

```
Display entries in HashMap
{Cook=29, Smith=30, Lewis=29, Anderson=31}

Display entries in ascending order of key
{Anderson=31, Cook=29, Lewis=29, Smith=30}
The age for Lewis is 29

Display entries in LinkedHashMap
{Smith=30, Anderson=31, Cook=29, Lewis=29}
```

As shown in the output, the entries in the **HashMap** are in random orde **TreeMap** are in increasing order of the keys. The entries in the **Linkec** order of their access, from least recently accessed to most recently.

All the concrete classes that implement the **Map** interface have at least tv is the no-arg constructor that constructs an empty map, and the other consı instance of **Map**. Thus **new TreeMap<String, Integer>(hashM** constructs a tree map from a hash map.

You can create an insertion-ordered or access-ordered linked hash map. An access-ordered linked hash map is created in lines 22–23. The most recently accessed entry is placed at the end of the map. The entry with the key Lewis is last accessed in line 31, so it is displayed last in line 34.

Tip

If you don't need to maintain an order in a map when updating it, use a **HashMap**. When you need to maintain the insertion order or access order in the map, use a **LinkedHashMap**. When you need the map to be sorted on keys, use a **TreeMap**.

22.11.1 Case Study: Occurrences of Words

This case study writes a program that counts the occurrences of words in a text and displays the words and their occurrences in alphabetical order of words. The program uses a **TreeMap** to store an entry consisting of a word and its count. For each word, check whether it is already a key in the map. If not, add to the map an entry with the word as the key and value **1**. Otherwise, increase the value for the word (key) by **1** in the map. Assume the words are case insensitive; e.g., **Good** is treated the same as **good**.

Listing 22.12 gives the solution to the problem.

LISTING 22.12 CountOccurrenceOfWords.java

```java
import java.util.*;

public class CountOccurrenceOfWords {
  public static void main(String[] args) {
    // Set text in a string
    String text = "Good morning. Have a good class. " +
      "Have a good visit. Have fun!";

    // Create a TreeMap to hold words as key and count as value
    TreeMap<String, Integer> map = new TreeMap<String, Integer>();

    String[] words = text.split("[ \n\t\r.,;:!?(){");
    for (int i = 0; i < words.length; i++) {
      String key = words[i].toLowerCase();

      if (key.length() > 0) {
        if (map.get(key) == null) {
          map.put(key, 1);
        }
        else {
          int value = map.get(key).intValue();
          value++;
          map.put(key, value);
        }
      }
    }

    // Get all entries into a set
    Set<Map.Entry<String, Integer>> entrySet = map.entrySet();

    // Get key and value from each entry
    for (Map.Entry<String, Integer> entry: entrySet)
      System.out.println(entry.getValue() + "\t" + entry.getKey());
  }
}
```

tree map
split string
add entry
add entry
tree map
entry set
display entry

```
2    a
1    class
1    fun
3    good
3    have
1    morning
1    visit
```

The program creates a **TreeMap** (line 10) to store pairs of words and their
The words serve as the keys. Since all elements in the map must be stc
count is wrapped in an **Integer** object.

The program extracts a word from a text using the **split** method (line
class (see §9.2.7). For each word extracted, the program checks whether it
a key in the map (line 17). If not, a new pair consisting of the word and its
stored to the map (line 18). Otherwise, the count for the word is increı
21–23).

The program obtains the entries of the map in a set (line 29), and trave
play the count and the key in each entry (lines 32–33).

Since the map is a tree map, the entries are displayed in increasing orde
play them in ascending order of the occurrence counts, see Exercise 22.8.

Now sit back and think how you would write this program without usi
program will be longer and more complex. You will find that **map** is a very
erful data structure for solving problems such as this.

22.12 Singleton and Unmodifiable Collectior and Maps

The **Collections** class contains the static methods for lists and collectio
the methods for creating singleton sets, lists, and maps, and for creating
lists, and maps, as shown in Figure 22.19.

The **Collections** class defines three constants: one for an empty set, or
and one for an empty map (**EMPTY_SET**, **EMPTY_LIST**, and **EMPTY_MAP**).
vides the **singleton(Object o)** method for creating an immutable set coı
gle item, the **singletonList(Object o)** method for creating an immut

[illegible]	
+singleton(o: Object): Set	Returns a singleton set containing the
+singletonList(o: Object): List	Returns a singleton list containing the
+singletonMap(key: Object, value: Object): Map	Returns a singleton map with the key a
+unmodifiedCollection(c: Collection): Collection	Returns an unmodified collection.
+unmodifiableList(list: List): List	Returns an unmodified list.
+unmodifiableMap(m: Map): Map	Returns an unmodified map.
+unmodifiableSet(s: Set): Set	Returns an unmodified set.
+unmodifiableSortedMap(s: SortedMap): SortedMap	Returns an unmodified sorted map.
+unmodifiableSortedSet(s: SortedSet): SortedSet	Returns an unmodified sorted set.

FIGURE 22.19 The **Collections** class contains the static methods for creating singleton and unmodif
and maps.