Sets and Maps

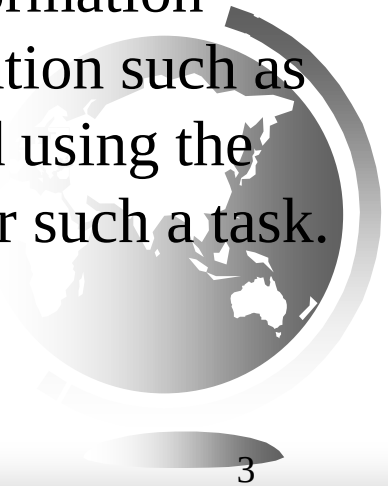
Objectives

- To store unordered, nonduplicate elements using a set (§21.2).
- To explore how and when to use HashSet (§21.2.1), LinkedHashSet (§21.2.2), or TreeSet (§21.2.3) to store elements.
- To compare performance of sets and lists (§21.3).
- To use sets to develop a program that counts the keywords in a Java source file (§21.4).
- To tell the differences between Collection and Map and describe when and how to use HashMap, LinkedHashMap, and TreeMap to store values associated with keys (§21.5).
- To use maps to develop a program that counts the occurrence of the words in a text (§21.6).
- To obtain singleton sets, lists, and maps, and unmodifiable sets, lists, and maps, using the static methods in the Collections class (§21.7).

Motivations

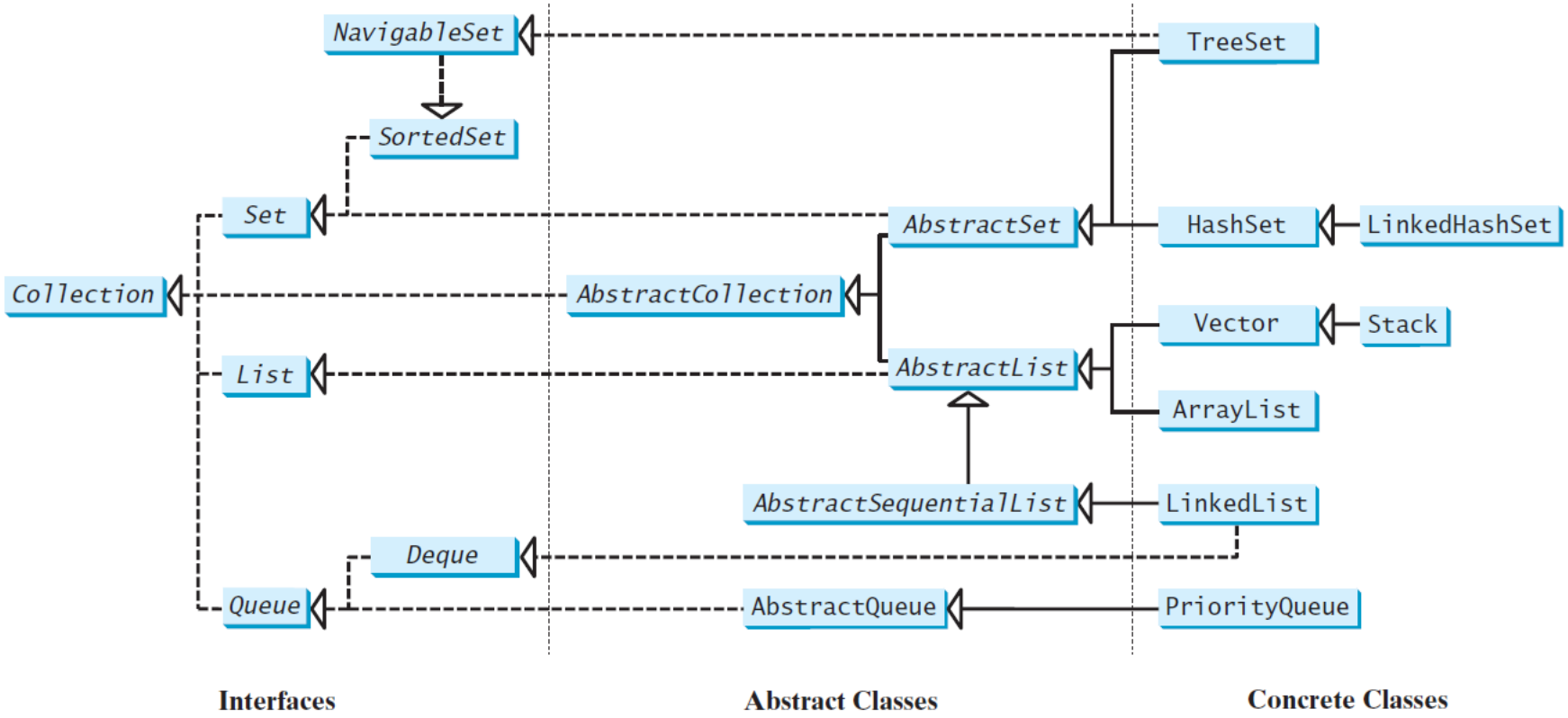
The "**No-Fly**" **list** is a list, created and maintained by the U.S. government's Terrorist Screening Center, of people who are not permitted to board a commercial aircraft for travel in or out of the United States. Suppose we need to write a program that checks whether a person is on the No-Fly list. You can use a list to store names in the No-Fly list. However, a more efficient data structure for this application is a *set*.

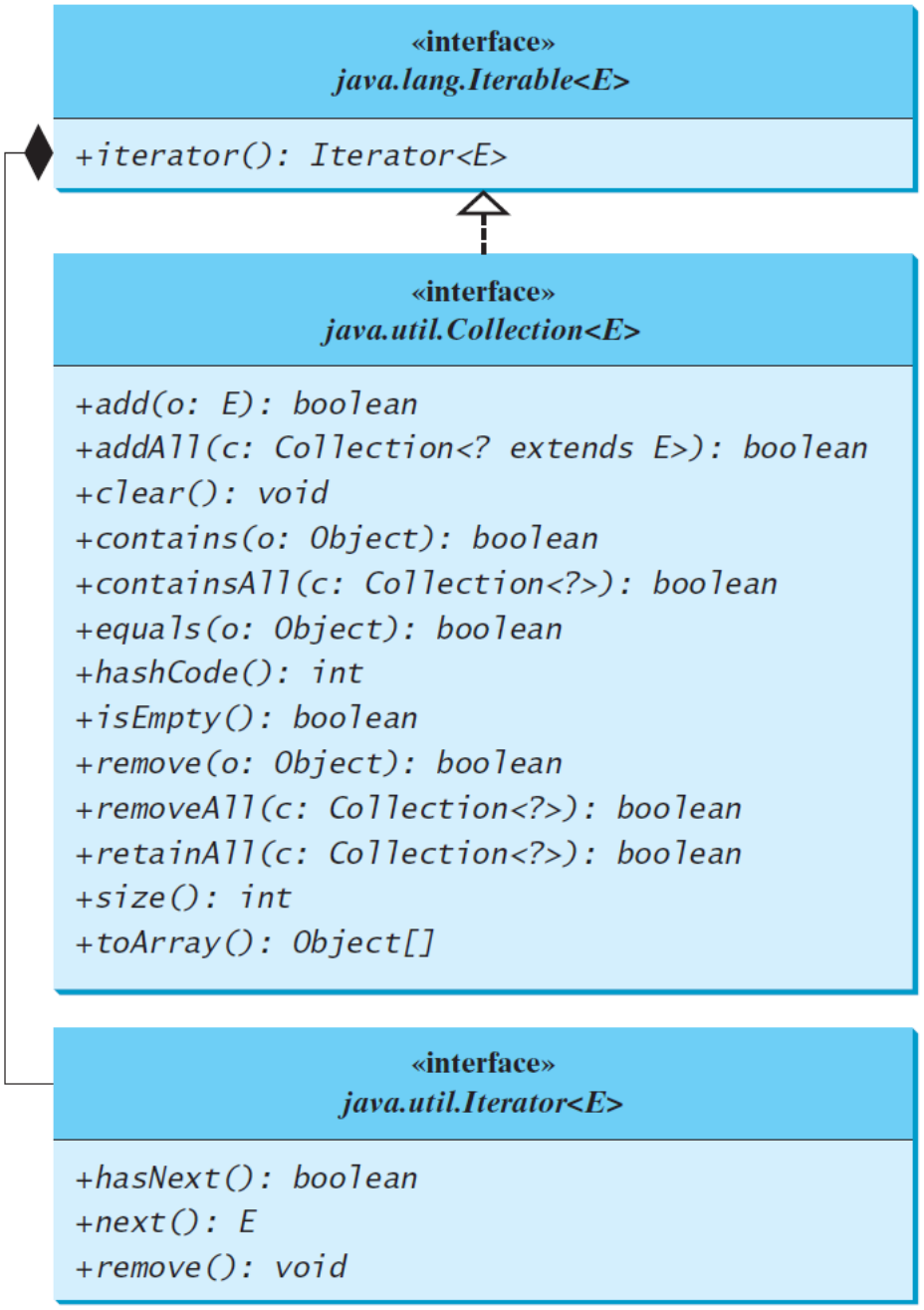
Suppose your program also needs to store detailed information about terrorists in the No-Fly list. The detailed information such as gender, height, weight, and nationality can be retrieved using the name as the key. A *map* is an efficient data structure for such a task.



Review of Java Collection Framework hierarchy

Set and List are subinterfaces of Collection.



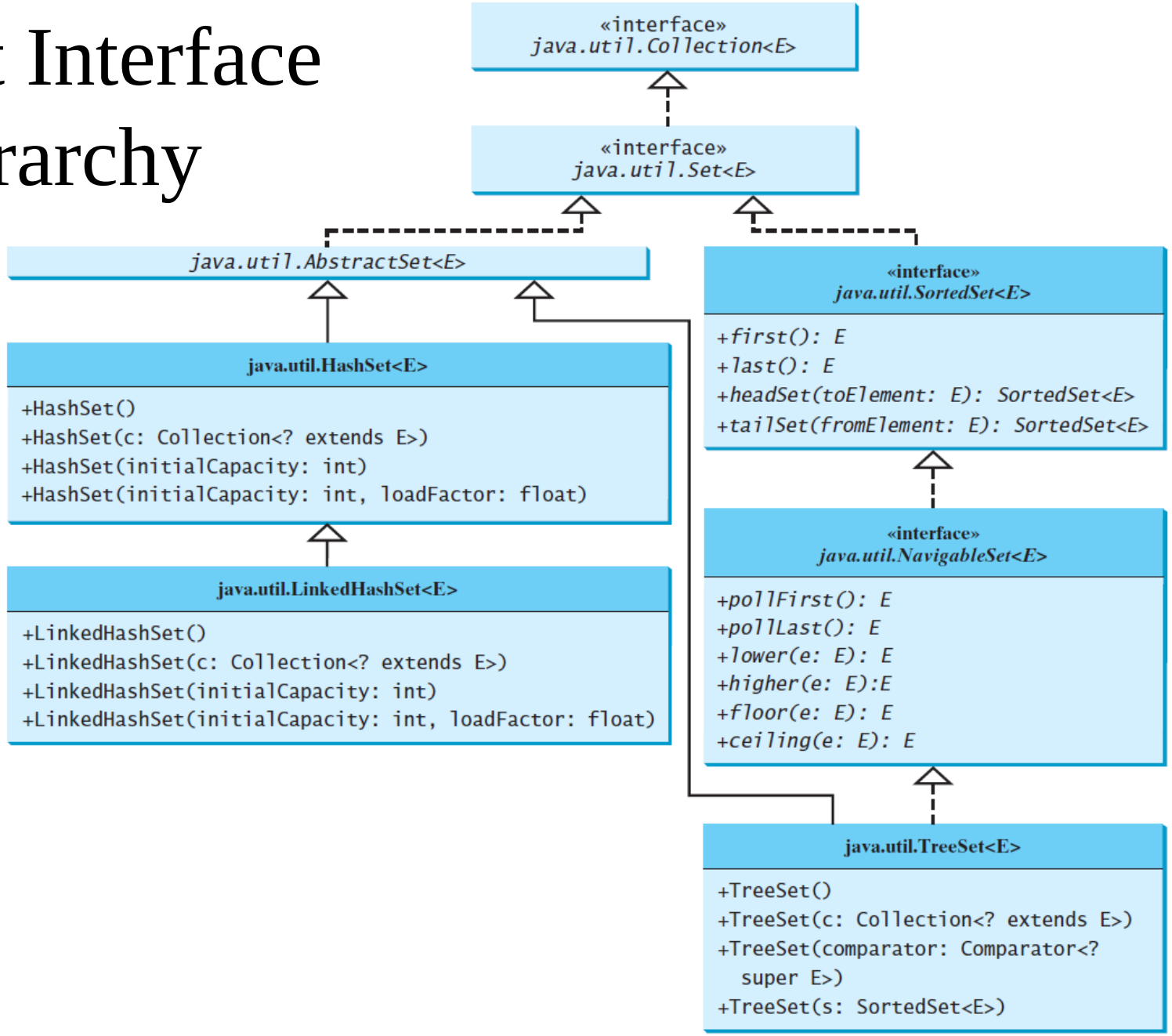


The Collection interface is the root interface for manipulating a collection of objects.

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

Run

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

Run

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

Run

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

Run

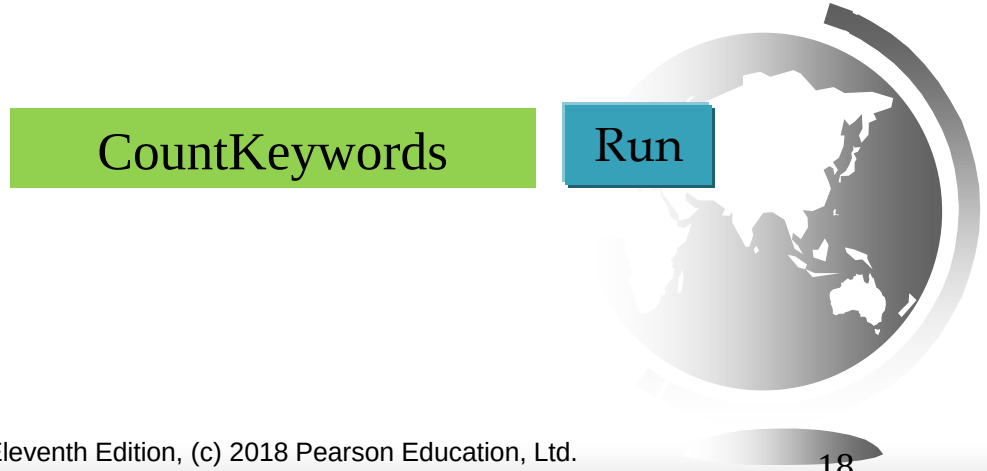
Performance of Sets and Lists

TestTreeSetWithComparator

Run

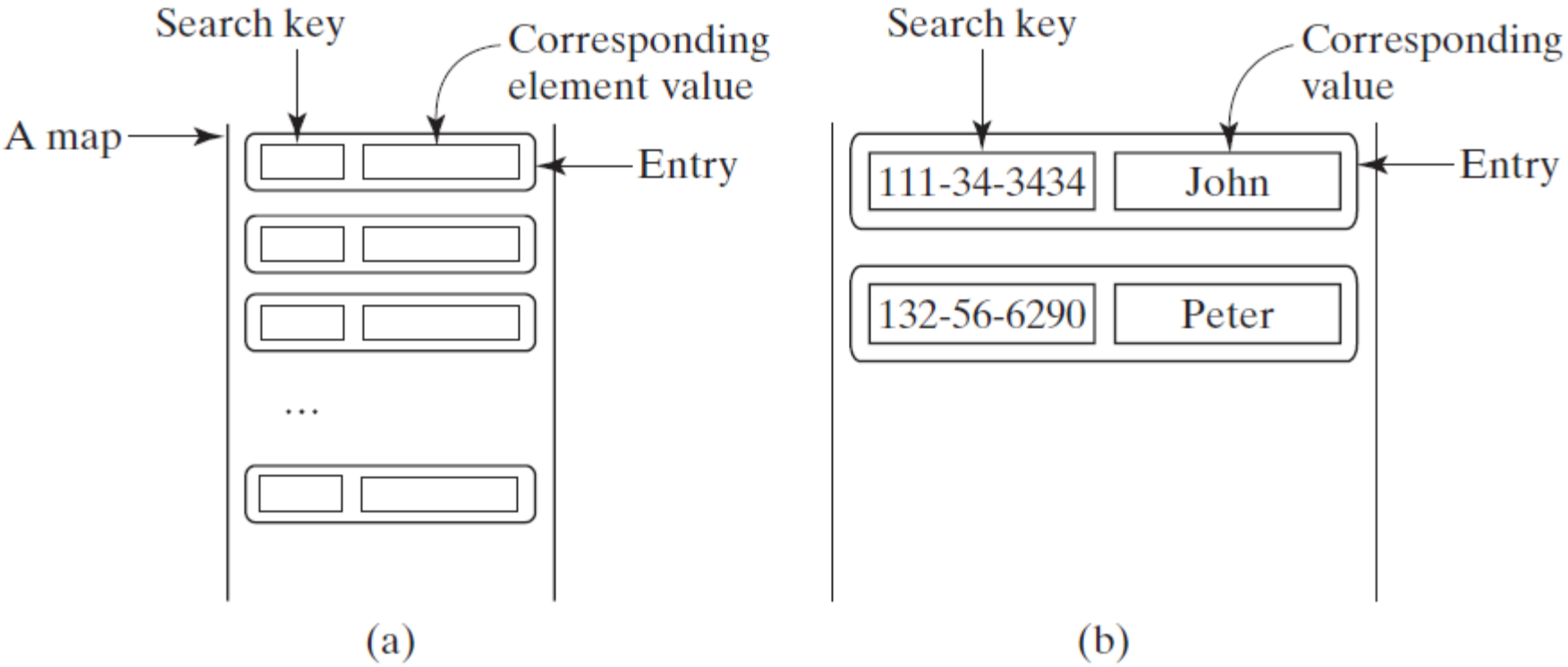
Case Study: Counting Keywords

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



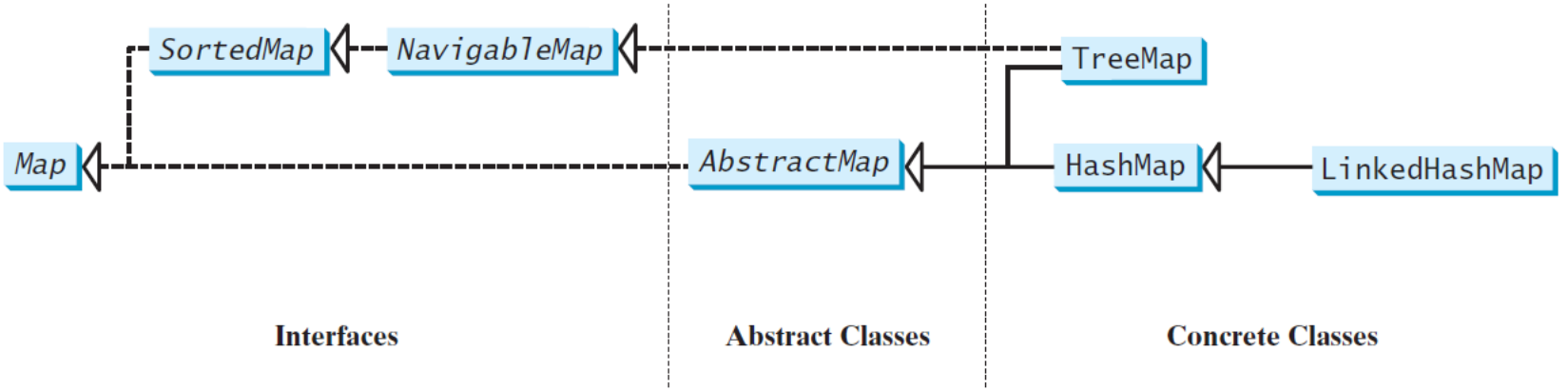
The Map Interface

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.

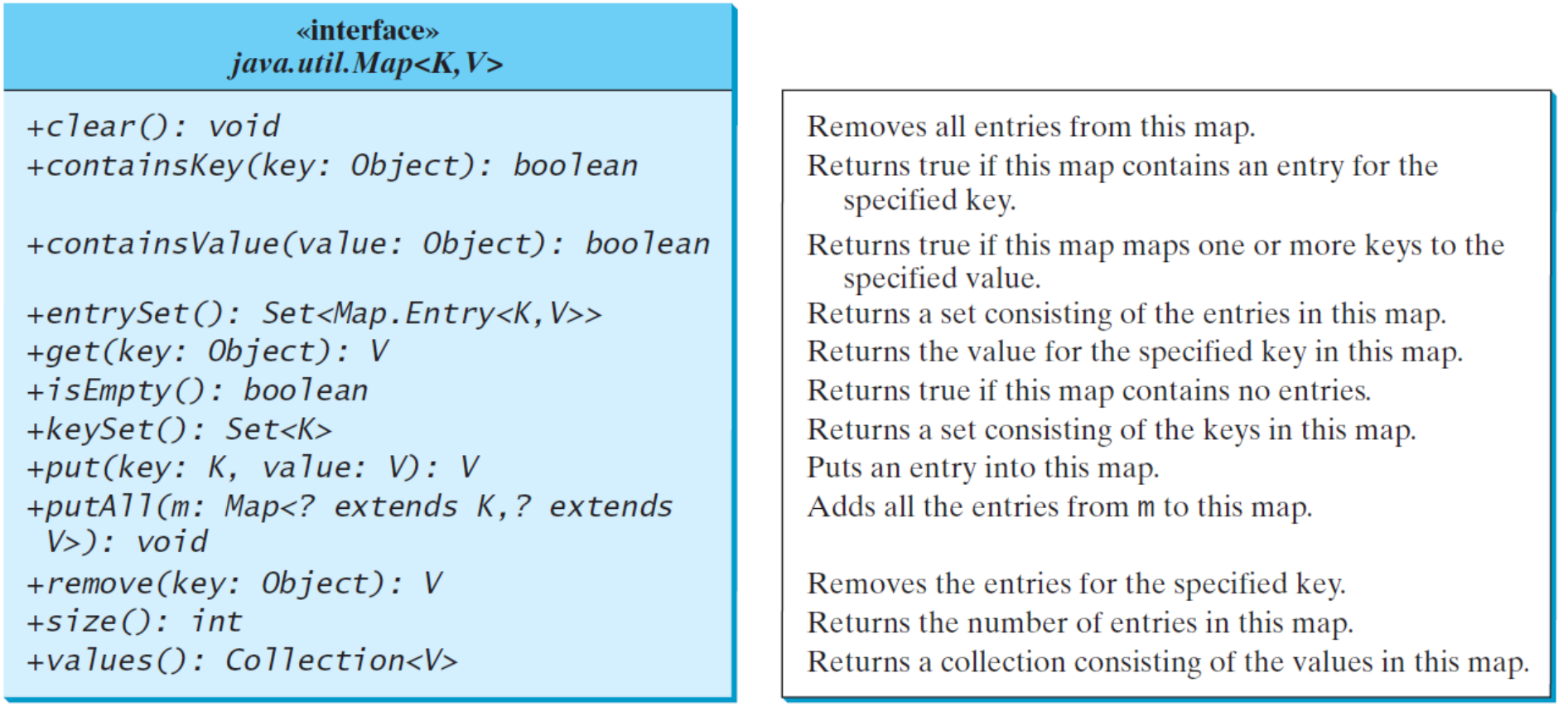


Map Interface and Class Hierarchy

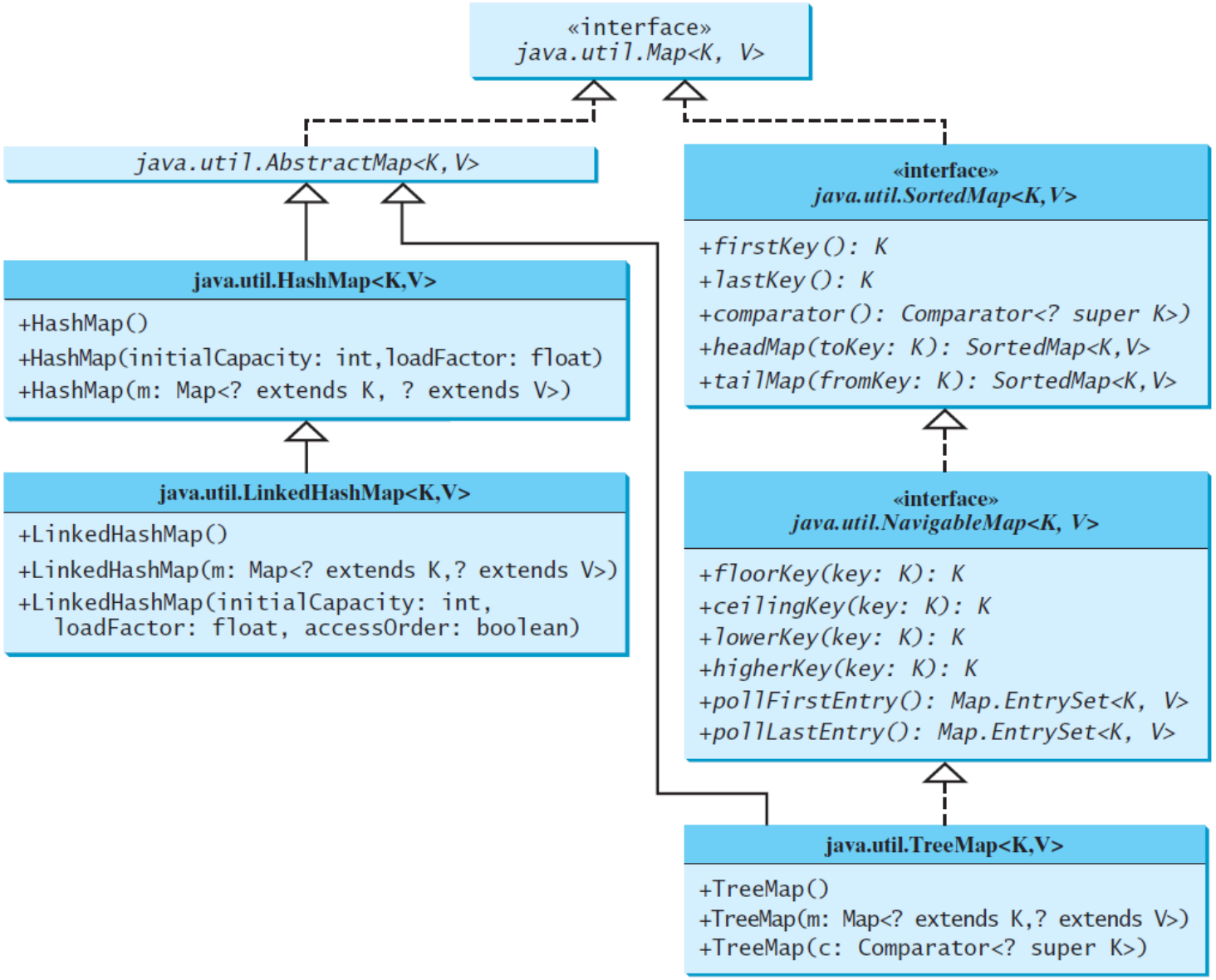
An instance of Map represents a group of objects, each of which is associated with a key. You can get the object from a map using a key, and you have to use a key to put the object into the map.



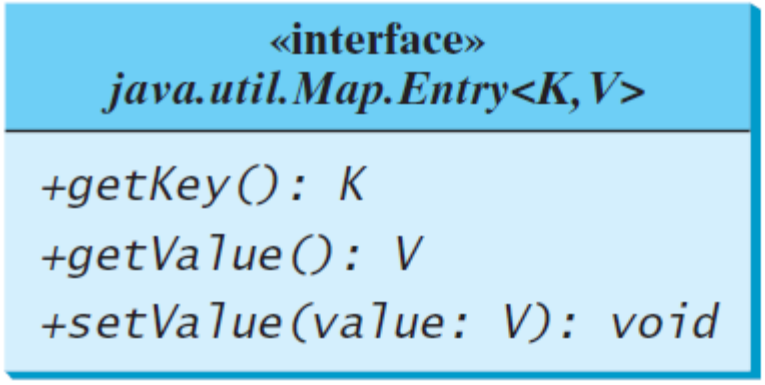
The Map Interface UML Diagram



Concrete Map Classes



Entry



`getKey()`	Returns the key from this entry.
`getValue()`	Returns the value from this entry.
`setValue(value: V)`	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

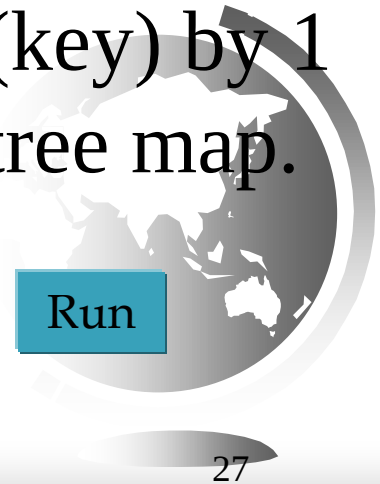
Run

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

Run



The Singleton and Unmodiable Collections

java.util.Collections	
+singleton(o: Object): Set	Returns an immutable set containing the specified object.
+singletonList(o: Object): List	Returns an immutable list containing the specified object.
+singletonMap(key: Object, value: Object): Map	Returns an immutable map with the key and value pair.
+unmodifiableCollection(c: Collection): Collection	Returns a read-only view of the collection.
+unmodifiableList(list: List): List	Returns a read-only view of the list.
+unmodifiableMap(m: Map): Map	Returns a read-only view of the map.
+unmodifiableSet(s: Set): Set	Returns a read-only view of the set.
+unmodifiableSortedMap(s: SortedMap): SortedMap	Returns a read-only view of the sorted map.
+unmodifiableSortedSet(s: SortedSet): SortedSet	Returns a read-only view of the sorted set.