# 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*.

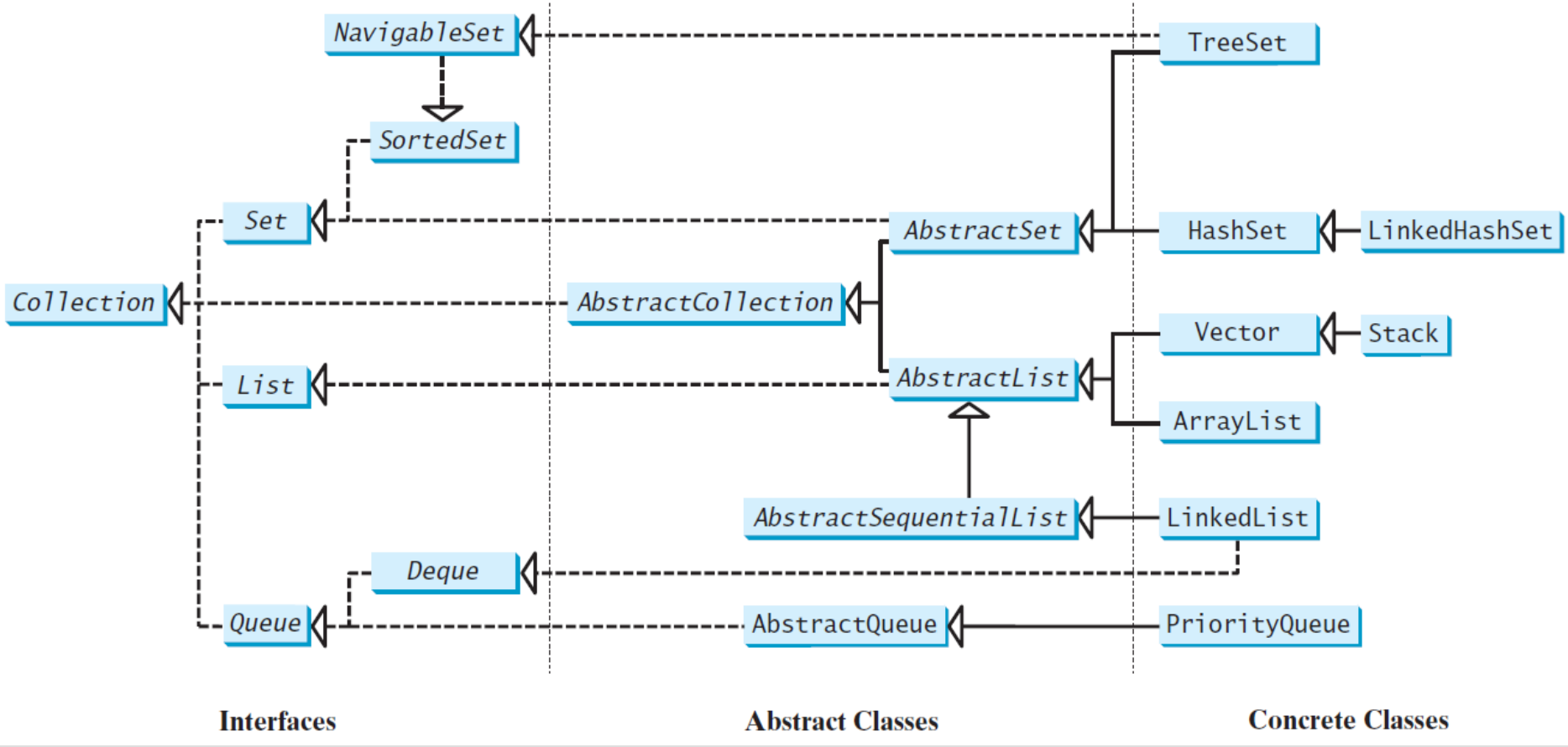
* Suppose your program also needs to store detailed information about terrorists in the No-Fly list.

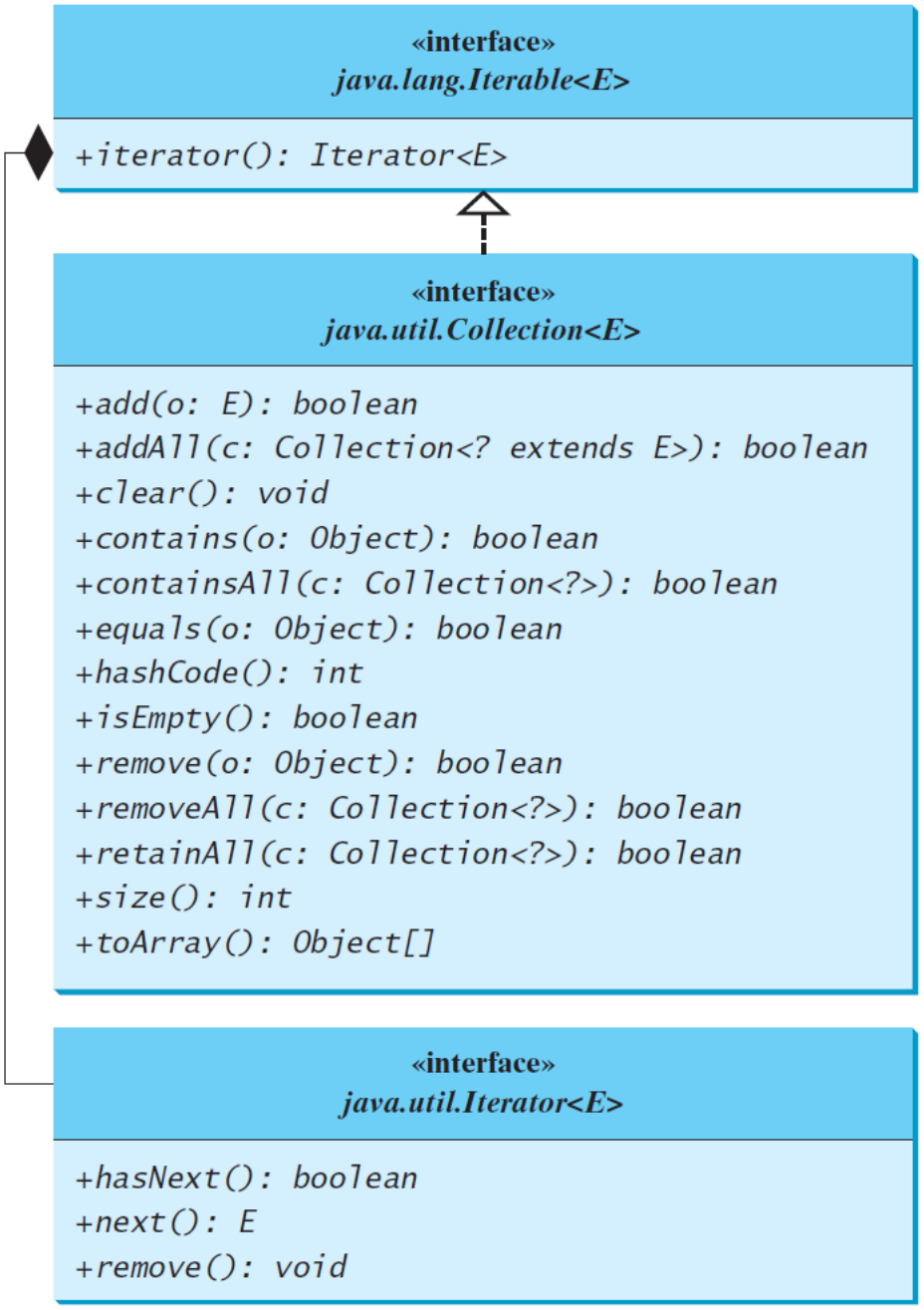
  - The detailed information such as gender, height, weight, and nationality can be retrieved using the name as the key.

  - 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

* About Set

- 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.

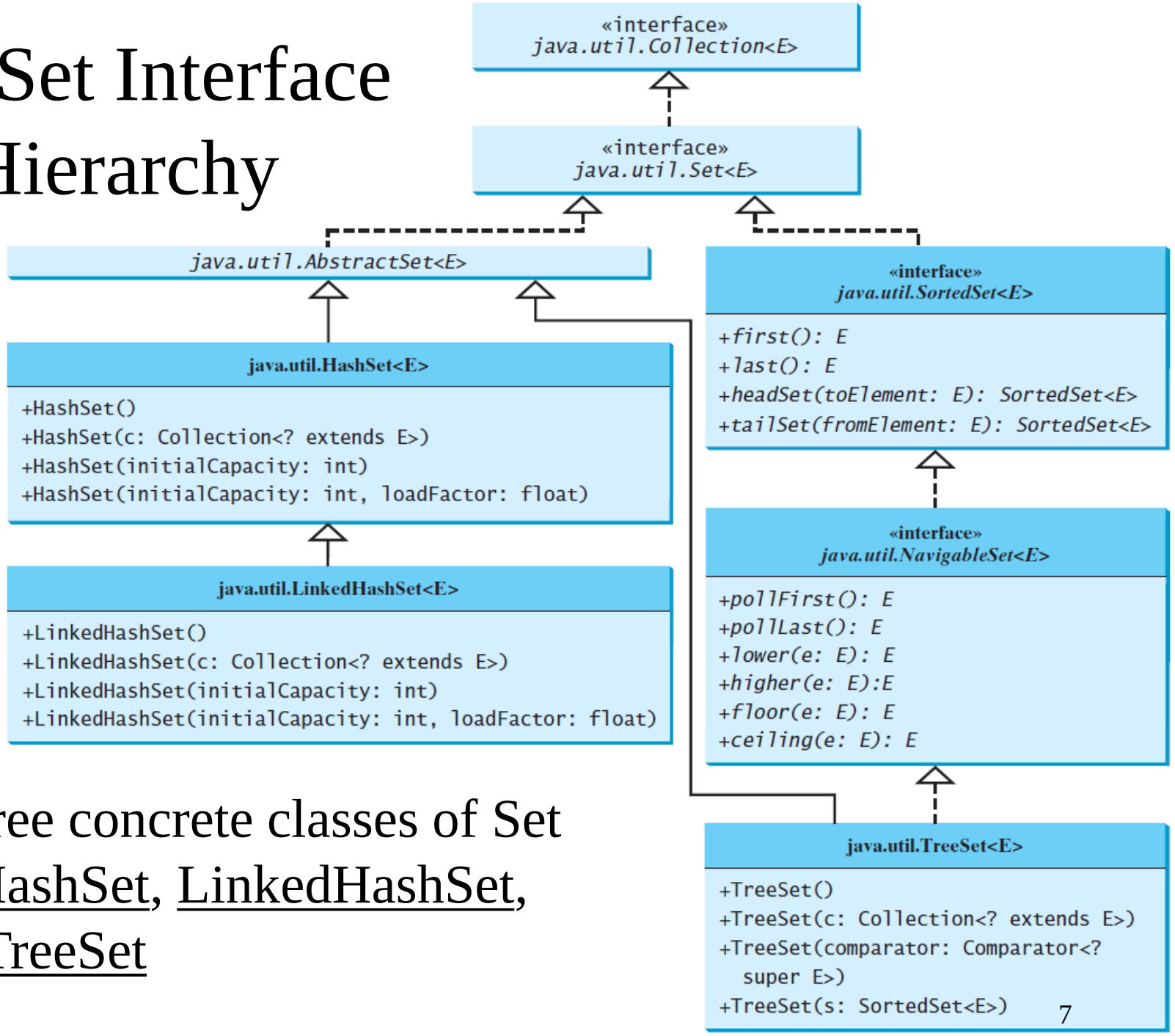
* About implementation

- The AbstractSet class extends AbstractCollection and partially implements Set.

- The AbstractSet class provides concrete implementations for the equals method and the hashCode method.

- Since the size method and iterator method are not implemented in the AbstractSet class, AbstractSet is an abstract class.

# The Set Interface Hierarchy




* Three concrete classes of Set are HashSet, LinkedHashSet, and TreeSet

# The HashSet Class

* The HashSet class is a concrete class that implements Set.

* By default, the initial capacity is 16 and load factor is 0.75.

  - If you know the size of your set, you can specify the initial capacity and load factor in the constructor. Otherwise, use the default setting.

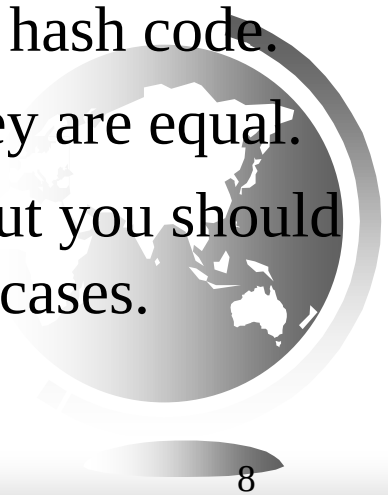
  - The load factor measures how full the set is allowed to be before its capacity is increased. The load factor is a value between 0.0 and 1.0.

* A HashSet 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.

  - The hash codes of two objects must be the same if they are equal.

  - Two unequal objects may have the same hash code, but you should implement the hashCode method to avoid too many such cases.



# LinkedHashSet

* LinkedHashSet extends HashSet with linked-list implementation that supports an ordering of the elements in the set.

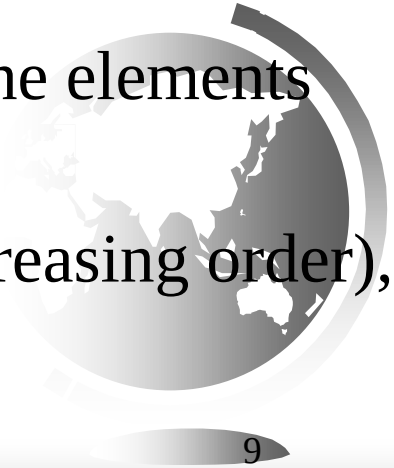
  - The elements in a HashSet are not ordered, but the elements in a LinkedHashSet can be retrieved in the order in which they were inserted into the set.

  - A LinkedHashSet can be created by using one of its four constructors.

  - These constructors are similar to the constructors for HashSet.

* The LinkedHashSet maintains the order in which the elements are inserted.

* To impose a different order (e.g., increasing or decreasing order), you can use the TreeSet class.



# TreeSet Class

* SortedSet is a subinterface of Set, which guarantees that the elements in the set are sorted.

  - Additionally, it provides the methods first() and last() for returning the first and last elements in the set,

  - and headSet(toElement) and tailSet(fromElement) for returning a portion of the set whose elements are less than toElement and greater than or equal to fromElement.

* TreeSet implements the SortedSet interface.

  - To create a TreeSet, use a constructor. You can add objects into a tree set as long as they can be compared with each other.

  - As discussed in Section 20.5, the elements can be compared in two ways: using the Comparable interface or the Comparator interface.

# Example: Using TreeSet to Sort Elements

* Listing 21.4 gives an example of ordering elements using the compareTo method in the Comparable interface.

* This example rewrites the preceding example to display the strings in alphabetical order using the TreeSet class.

* If you create a TreeSet using its no-arg constructor, the compareTo method is used to compare the elements in the set, assuming that the class of the elements implements the Comparable interface.

* To use a comparator, you can use the constructor TreeSet(Comparator comparator) to create a sorted set that uses the compare method in the comparator to order the elements in the set.

# Performance of Sets and Lists

* The elements in a list can be accessed through the index.

* However, sets do not support indexing, because the elements in a set are unordered.

* To traverse all elements in a set, use a foreach loop.

* We now conduct an interesting experiment to test the performance of sets and lists.

- Listing 21.6 gives a program that shows the execution time of

(1) testing whether an element is in a hash set, linked hash set, tree set, array list, and linked list,

(2) removing elements from a hash set, linked hash set, tree set, array list, and linked list.

- As these runtimes illustrate, sets are much more efficient than lists for testing whether an element is in a set or a list.

# Case Study: Counting Keywords

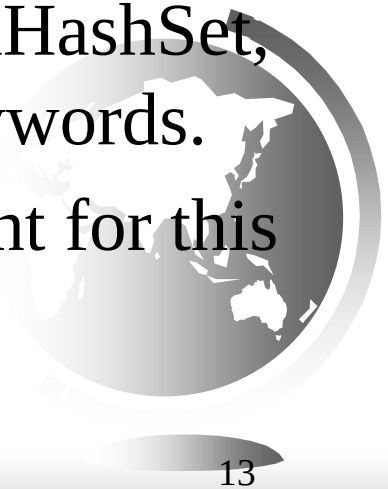
* This section presents an application that counts the number of the keywords in a Java source file.

- For each word in a Java source file, we need to determine whether the word is a keyword.

- To handle this efficiently, store all the keywords in a HashSet and use the contains method to test if a word is in the keyword set.

- You may rewrite the program to use a LinkedHashSet, TreeSet, ArrayList, or LinkedList to store the keywords.

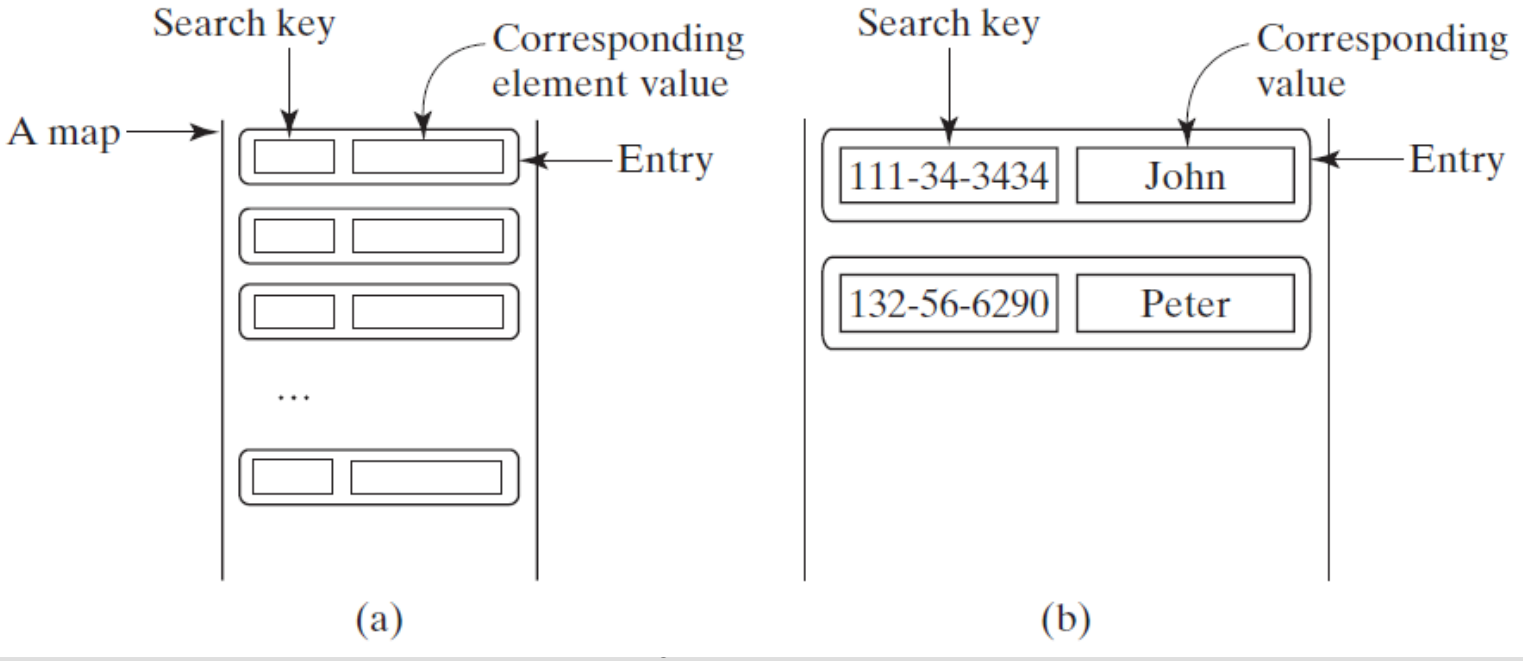
- However, using a HashSet is the most efficient for this program.



# The Map Interface

* A map is a container object that stores a collection of key/value pairs.

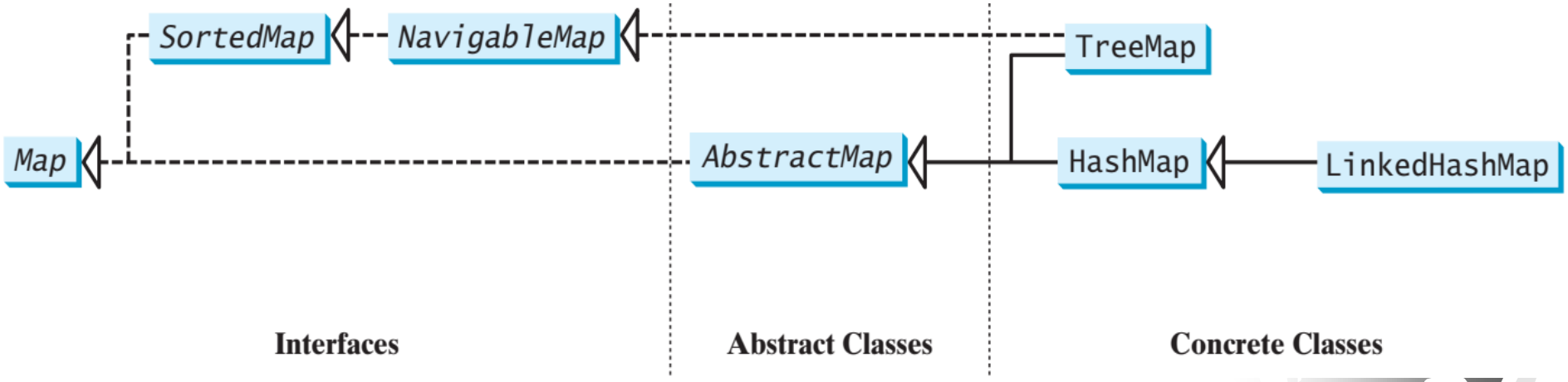
- It enables fast retrieval, deletion, and updating of the pair through the key.
- A map stores the values along with the keys. The keys are like indexes.
- In List, the indexes are integers. In Map, the keys can be any objects.
- A map cannot contain duplicate keys. Each key maps to one value.



# Map Interface and Class Hierarchy

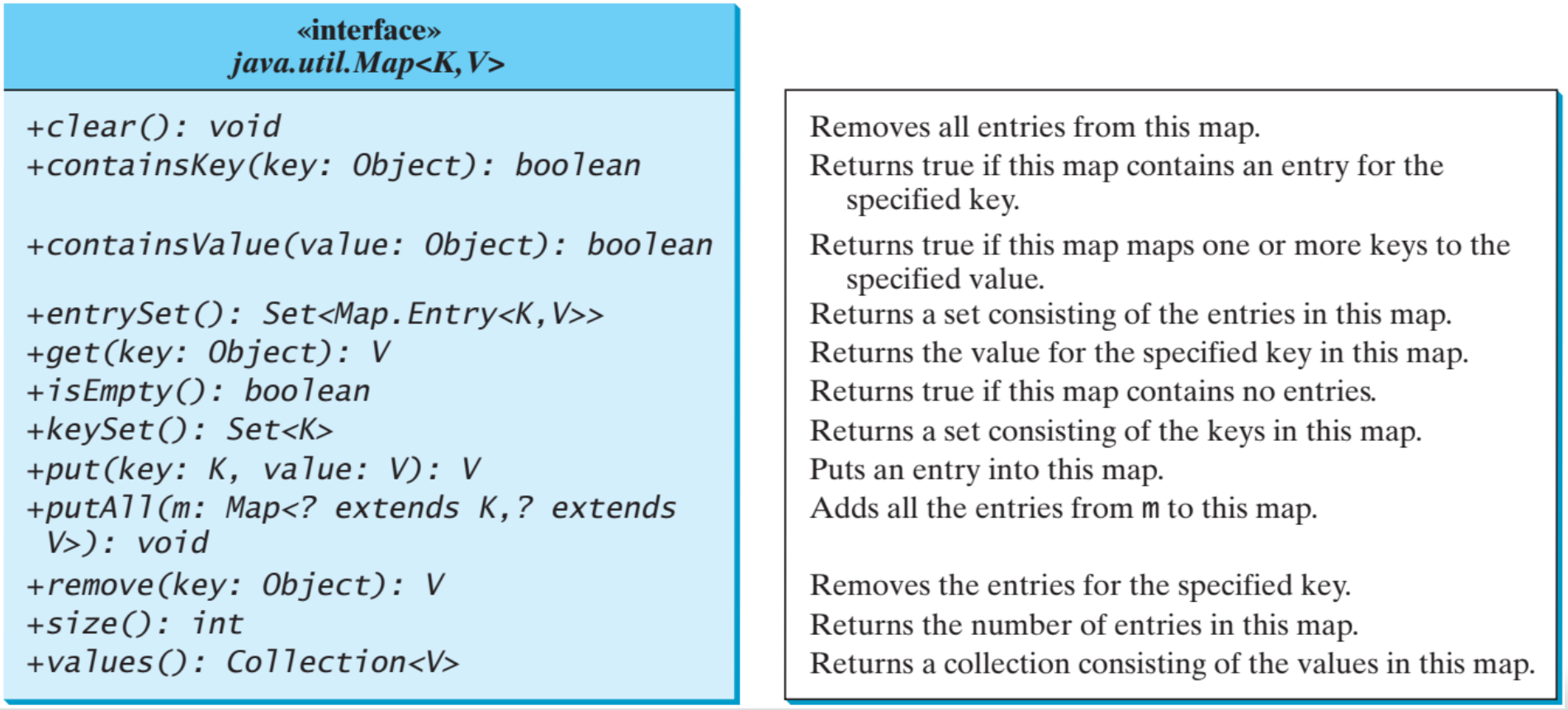
* There are three types of maps: HashMap, LinkedHashMap, and TreeMap.

* The common features of these maps are defined in the Map interface.

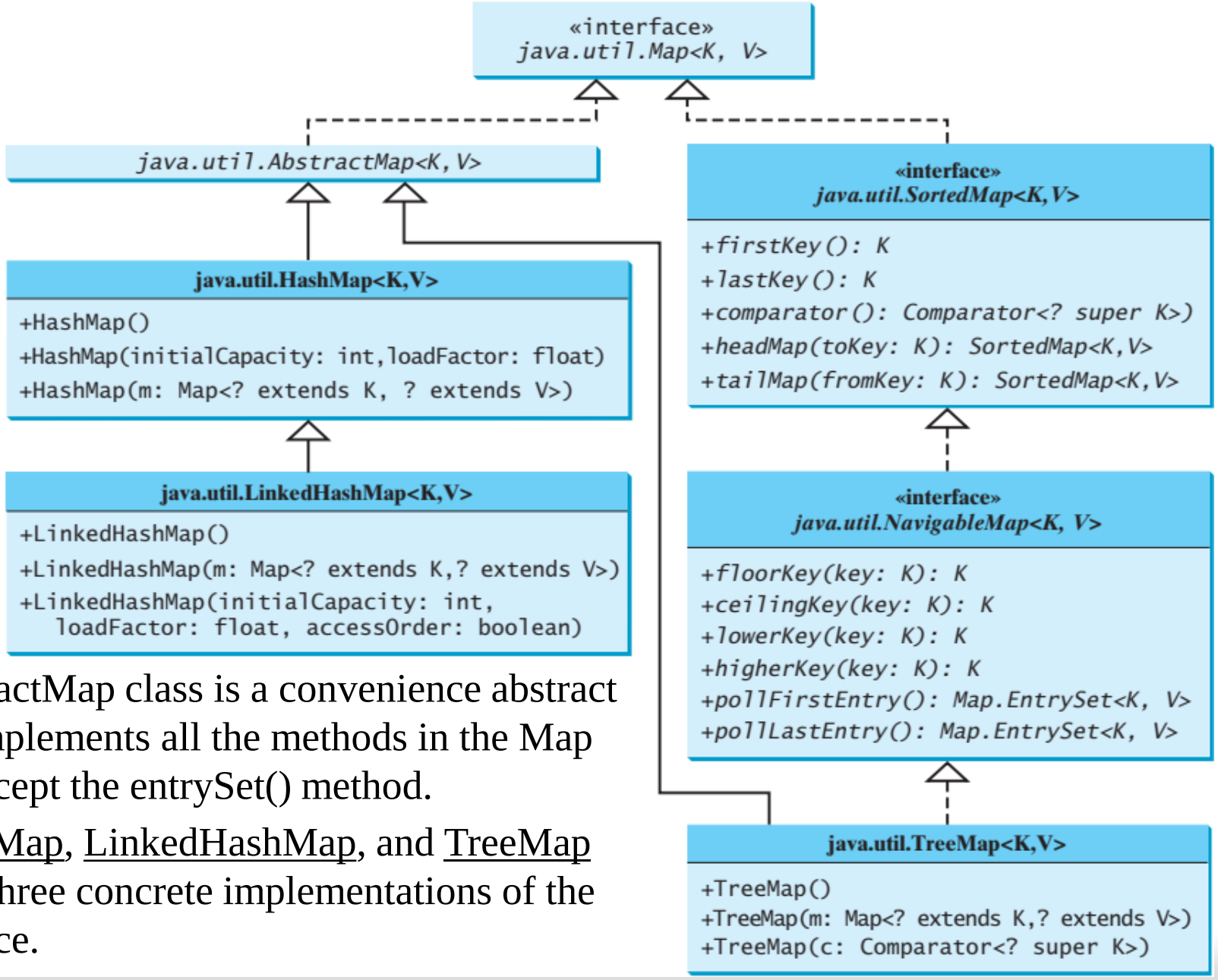


# The Map Interface UML Diagram

* The Map interface provides the methods for querying, updating, and obtaining a collection of values and a set of keys.



# Concrete Map Classes




* The AbstractMap class is a convenience abstract class that implements all the methods in the Map interface except the entrySet() method.

* The HashMap, LinkedHashMap, and TreeMap classes are three concrete implementations of the Map interface.

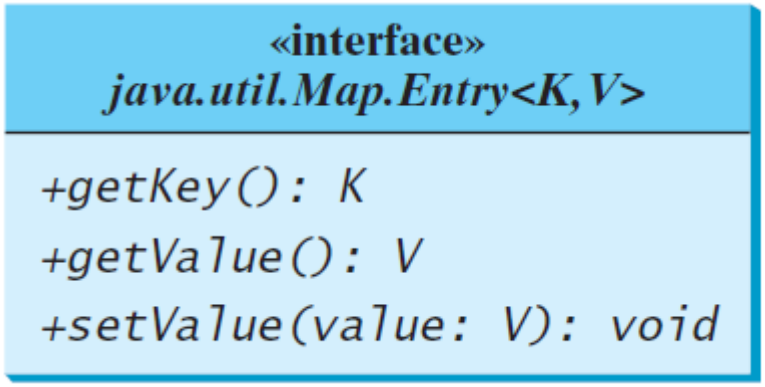
# Entry

* You can obtain a set of the keys in the map using the keySet() method, and a collection of the values in the map using the values() method.

* The entrySet() method returns a set of entries.

- The entries are instances of the Map.Entry<K, V> interface, where Entry is an inner interface for the Map interface.

- Each entry in the set is a key/value pair in the underlying map.




| |
|---|
| Returns the key from this entry. |
| Returns the value from this entry. |
| Replaces the value in this entry with a new value. |

# HashMap and LinkedHashMap

* The HashMap class is efficient for locating a value, inserting an entry, and deleting an entry.

* LinkedHashMap extends HashMap with a linked-list implementation that supports an ordering of the entries.

* Ordering

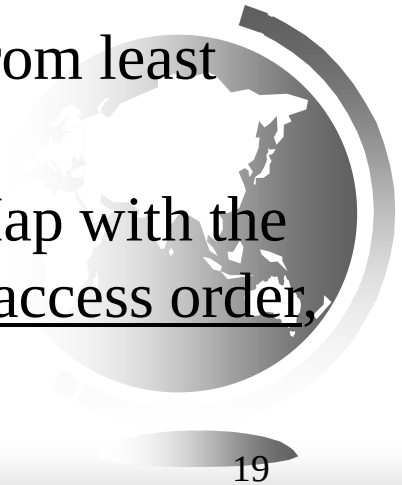
  - The entries in a HashMap are not ordered,

  - but the entries in a LinkedHashMap can be retrieved

    • either in the order in which they were inserted into the map (known as the insertion order)

    • or in the order in which they were last accessed, from least recently to most recently accessed (access order).

    • The no-arg constructor constructs a LinkedHashMap with the insertion order. To construct a LinkedHashMap with the access order, use LinkedHashMap(initialCapacity, loadFactor, true).



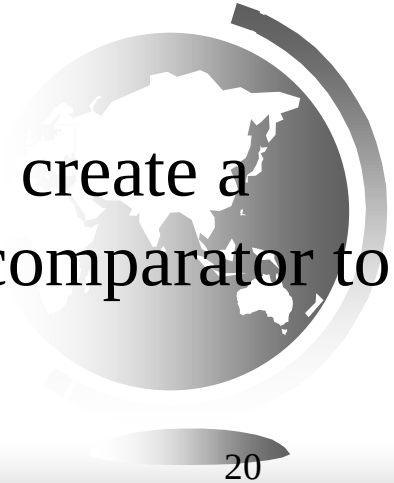
# TreeMap

* The TreeMap class is efficient for traversing the keys in a sorted order.

- The keys can be sorted using the Comparable interface or the Comparator interface.

- If you create a TreeMap using its no-arg constructor, the compareTo method in the Comparable interface is used to compare the keys in the map, assuming that the class for the keys implements the Comparable interface.

- To use a comparator, you have to use the TreeMap(Comparator comparator) constructor to create a sorted map that uses the compare method in the comparator to order the entries in the map based on the keys.



# SortedMap and NavigableMap

* SortedMap is a subinterface of Map, which guarantees that the entries in the map are sorted.

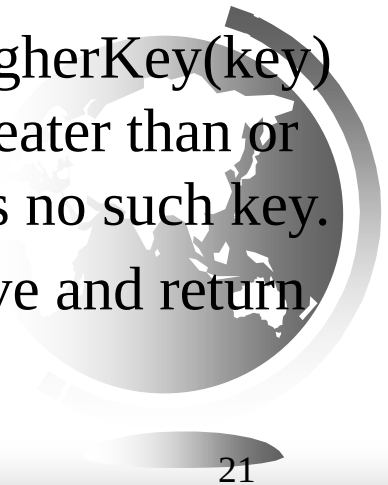
  - Additionally, it provides the methods firstKey() and lastKey() for returning the first and last keys in the map,

  - and headMap(toKey) and tailMap(fromKey) for returning a portion of the map whose keys are less than toKey and greater than or equal to fromKey, respectively.

* NavigableMap extends SortedMap to provide the 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.



# Example: Using HashMap, LinkedHashMap and TreeMap

* Listing 21.8 gives an example that creates a hash map, a linked hash map, and a tree map for mapping 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 entries 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.

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

* Writes a program that counts the occurrences of words in a text and displays the words and their occurrences in alphabetical order of the 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 an entry to the map 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.

# The Singleton and Unmodiable Collections

* The Collections class contains the static methods for lists and collections.

* It also contains the methods for creating immutable singleton sets, lists, and maps, and for creating read-only sets, lists, and maps, as shown in Figure 21.7.

| 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. |

# The Singleton and Unmodiable Collections, cont.

* The Collections class defines three constants—

  - EMPTY_SET for  empty set, EMPTY_LIST for empty list, EMPTY_MAP for empty map

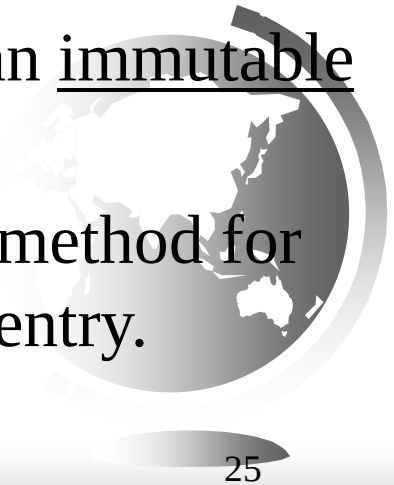
  - These collections are immutable.

* The class also provides,

  - the singleton(Object o) method for creating an immutable set containing only a single item,

  - the singletonList(Object o) method for creating an immutable list containing only a single item,

  - and the singletonMap(Object key, Object value) method for creating an immutable map containing only a single entry.



# The Singleton and Unmodiable Collections, cont.

* The Collections class also provides six static methods for returning read-only views for collections:

  unmodifiableCollection(Collection c),

  unmodifiableList(List list),

  unmodifiableMap(Map m),

  unmodifiableSet(Set set),

  unmodifiableSortedMap(SortedMap m),

  and unmodifiableSortedSet(Sorted Set s).

* This type of view is like a reference to the actual collection.

  - But you cannot modify the collection through a read-only view.

  - Attempting to modify a collection through a readonly view will cause an UnsupportedOperationException.