CHAPTER

21

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 a set of elements.
- To compare the 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`, or `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, use the static methods in the `Collections` class (§21.7).

21.1 Introduction

Key Point

A set is an efficient data structure for storing and processing nonduplicate elements. A map is like a dictionary that provides a quick lookup to retrieve a value using a key.

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*.

why 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.

why map?

This chapter introduces sets and maps in the Java Collections Framework.

21.2 Sets

Key Point

You can create a set using one of its three concrete classes: `HashSet`, `LinkedHashSet`, *or* `TreeSet`.

set
no duplicates

The `Set` interface extends the `Collection` interface, as shown in Figure 20.1. 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.

`AbstractSet`

The `AbstractSet` class extends `AbstractCollection` and partially 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 codes 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.

Three concrete classes of `Set` are `HashSet`, `LinkedHashSet`, and `TreeSet`, as shown in Figure 21.1.

21.2.1 HashSet

`hash set`

The `HashSet` class is a concrete class that implements `Set`. You can create an empty *hash set* using its no-arg constructor, or create a hash set from an existing collection. By default, the initial capacity is `16` and the 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 is a value between `0.0` and `1.0`.

load factor

The load factor measures how full the set is allowed to be before its capacity is increased. When the number of elements exceeds the product of the capacity and load factor, the capacity is automatically doubled. For example, if the capacity is `16` and load factor is `0.75`, the capacity will be doubled to `32` when the size reaches `12` ($16 * 0.75 = 12$). A higher load factor decreases the space costs but increases the search time. Generally, the default load factor `0.75` is a good trade-off between time and space costs. We will discuss more on the load factor in Chapter 27, Hashing.

`hashCode()`

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 `hashCode` method is defined in the `Object` class. The hash codes of two objects must be the same if the two objects are equal. Two unequal objects may have the same hash code, but you should implement the `hashCode` method to avoid too many such cases. Most of the classes in the Java API implement the `hashCode` method. For example, the `hashCode` in the `Integer` class returns its `int` value. The `hashCode` in the `Character` class returns the Unicode of the character. The `hashCode` in the `String` class returns $s_0 * 31^{(n-1)} + s_1 * 31^{(n-2)} + \cdots + s_{n-1}$, where s_i is `s.charAt(i)`.

`Set` does not store duplicate elements. Two elements `e1` and `e2` are considered duplicate for a `HashSet` if `e1.equals(e2)` is true and `e1.hashCode() == e2.hashCode()`. Note that

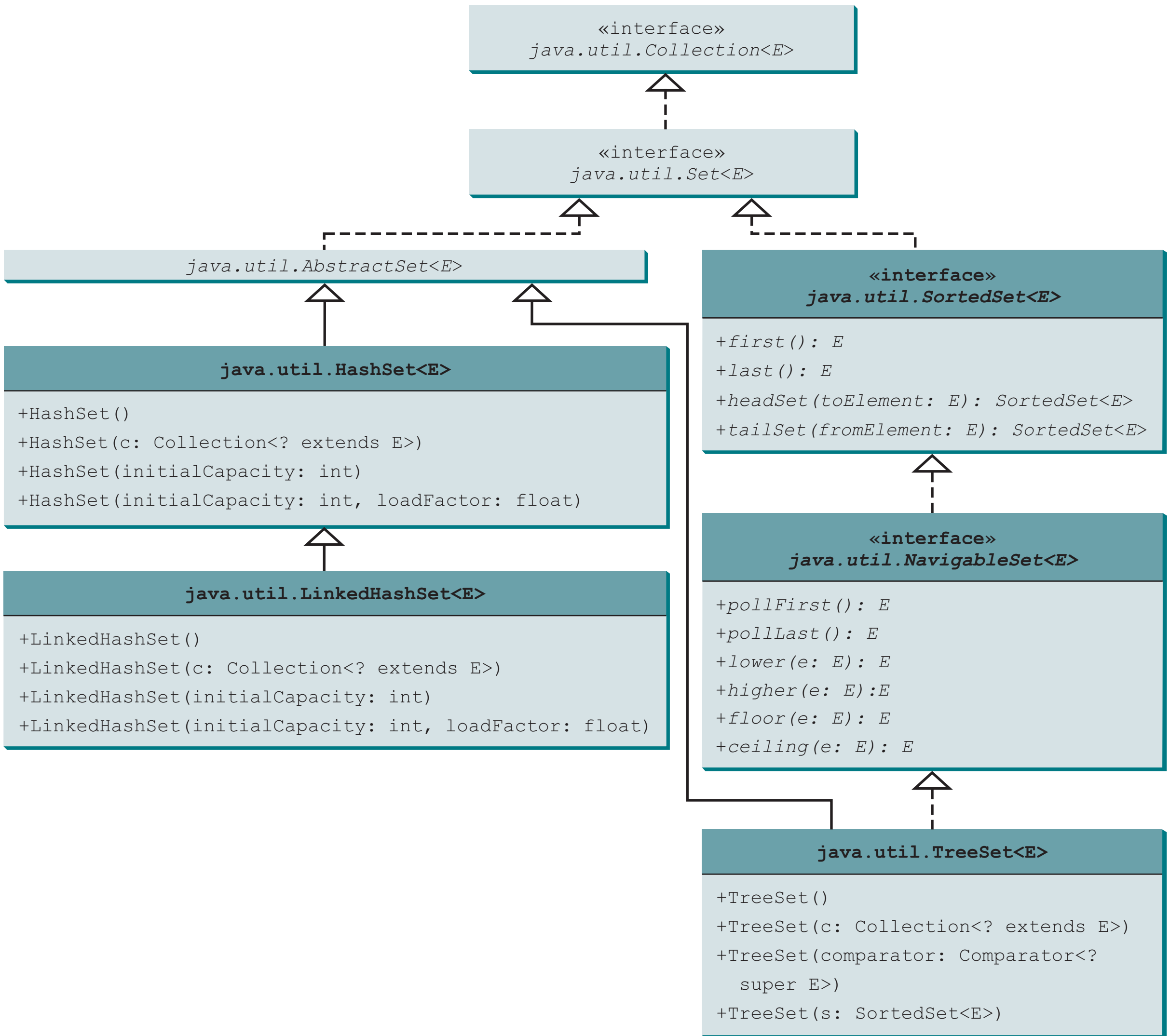

FIGURE 21.1 The Java Collections Framework provides three concrete set classes.

by contract, if two elements are equal, their `hashCode` must be same. So you need to override the `hashCode()` method whenever the `equals` method is overridden in the class.

Listing 21.1 gives a program that creates a hash set to store strings and uses a foreach loop and a `forEach` method to traverse the elements in the set.

Listing 21.1 gives a program that creates a hash set to store strings and uses a foreach loop and a `forEach` method to traverse the elements in the set.

LISTING 21.1 TestHashSet.java

```java
import java.util.*;

public class TestHashSet {
  public static void main(String[] args) {
    // Create a hash set
    Set<String> set = new HashSet<>();

    // Add strings to the set
    set.add("London");
```

create a set

add element

traverse elements

forEach method

```
10      set.add("Paris");
11      set.add("New York");
12      set.add("San Francisco");
13      set.add("Beijing");
14      set.add("New York");
15
16      System.out.println(set);
17
18      // Display the elements in the hash set
19      for (String s: set) {
20        System.out.print(s.toUpperCase() + " ");
21      }
22
23      // Process the elements using a forEach method
24      System.out.println();
25      set.forEach(e -> System.out.print(e.toLowerCase() + " "));
26    }
27  }
```

```
[San Francisco, New York, Paris, Beijing, London]
SAN FRANCISCO NEW YORK PARIS BEIJING LONDON
```

The strings are added to the set (lines 9–14). `New York` is added to the set more than once, but only one string is stored because a set does not allow duplicates.

As shown in the output, the strings are not stored in the order in which they are inserted into the set. There is no particular order for the elements in a hash set. To impose an order on them, you need to use the `LinkedHashSet` class, which is introduced in the next section.

Recall that the `Collection` interface extends the `Iterable` interface, so the elements in a set are iterable. A foreach loop is used to traverse all the elements in the set (lines 19–21). You can also use a `forEach` method to process each element in a set (line 25).

Since a set is an instance of `Collection`, all methods defined in `Collection` can be used for sets. Listing 21.2 gives an example that applies the methods in the `Collection` interface on sets.

Listing 21.2 TestMethodsInCollection.java

create a set

add element

get size

remove element

create a set

```
public class TestMethodsInCollection {
  public static void main(String[] args) {
    // Create set1
    java.util.Set<String> set1 = new java.util.HashSet<>();

    // Add strings to set1
    set1.add("London");
    set1.add("Paris");
    set1.add("New York");
    set1.add("San Francisco");
    set1.add("Beijing");

    System.out.println("set1 is " + set1);
    System.out.println(set1.size() + " elements in set1");

    // Delete a string from set1
    set1.remove("London");
    System.out.println("\nset1 is " + set1);
    System.out.println(set1.size() + " elements in set1");

    // Create set2
    java.util.Set<String> set2 = new java.util.HashSet<>();

    // Add strings to set2
```

```
25        set2.add("London");
26        set2.add("Shanghai");
27        set2.add("Paris");
28        System.out.println("\nset2 is " + set2);
29        System.out.println(set2.size() + " elements in set2");
30
31        System.out.println("\nIs Taipei in set2? "
32          + set2.contains("Taipei"));
33
34        set1.addAll(set2);
35        System.out.println("\nAfter adding set2 to set1, set1 is "
36          + set1);
37
38        set1.removeAll(set2);
39        System.out.println("After removing set2 from set1, set1 is "
40          + set1);
41
42        set1.retainAll(set2);
43        System.out.println("After retaining common elements in set2 "
44          + "and set2, set1 is " + set1);
45      }
46    }
```

add element

contains element?

`addAll`

`removeAll`

`retainAll`

```
set1 is [San Francisco, New York, Paris, Beijing, London]

5 elements in set1

set1 is [San Francisco, New York, Paris, Beijing]
4 elements in set1

set2 is [Shanghai, Paris, London]
3 elements in set2

Is Taipei in set2? false

After adding set2 to set1, set1 is
  [San Francisco, New York, Shanghai, Paris, Beijing, London]

After removing set2 from set1, set1 is
  [San Francisco, New York, Beijing]

After retaining common elements in set1 and set2, set1 is []
```

The program creates two sets (lines 4 and 22). The `size()` method returns the number of the elements in a set (line 14). Line 17

```
set1.remove("London");
```

removes `London` from `set1`.

The `contains` method (line 32) checks whether an element is in the set.

Line 34

```
set1.addAll(set2);
```

adds `set2` to `set1`. Therefore, `set1` becomes `[San Francisco, New York, Shanghai, Paris, Beijing, London]`.

Line 38

```
set1.removeAll(set2);
```

removes set2 from set1. Thus, set1 becomes [San Francisco, New York, Beijing].

Line 42

```
set1.retainAll(set2);
```

retains the common elements in set1 and set2. Since set1 and set2 have no common elements, set1 becomes empty.

21.2.2 LinkedHashSet

LinkedHashSet

LinkedHashSet extends HashSet with a 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, as shown in Figure 21.1. These constructors are similar to the constructors for HashSet.

Listing 21.3 gives a test program for LinkedHashSet. The program simply replaces HashSet by LinkedHashSet in Listing 21.1.

LISTING 21.3 TestLinkedHashSet.java

create linked hash set

add element

display elements

```
 1 import java.util.*;
 2
 3 public class TestLinkedHashSet {
 4   public static void main(String[] args) {
 5     // Create a hash set
 6     Set<String> set = new LinkedHashSet<>();
 7
 8     // Add strings to the set
 9     set.add("London");
10     set.add("Paris");
11     set.add("New York");
12     set.add("San Francisco");
13     set.add("Beijing");
14     set.add("New York");
15
16     System.out.println(set);
17
18     // Display the elements in the hash set
19     for (String element: set)
20       System.out.print(element.toLowerCase() + " ");
21   }
22 }
```

```
[London, Paris, New York, San Francisco, Beijing]
london paris new york san francisco beijing
```

A LinkedHashSet is created in line 6. As shown in the output, the strings are stored in the order in which they are inserted. Since LinkedHashSet is a set, it does not store duplicate elements.

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, which is introduced in the next section.

Tip

If you don't need to maintain the order in which the elements are inserted, use HashSet, which is more efficient than LinkedHashSet.

21.2.3 TreeSet

As shown in Figure 21.1, **SortedSet** is a subinterface of **Set**, which guarantees that the elements in the set are sorted. In addition, 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**, respectively.

NavigableSet extends **SortedSet** to provide navigation methods **lower(e)**, **floor(e)**, **ceiling(e)**, and **higher(e)** that return elements, respectively, less than, less than or equal, greater than or equal, and greater than a given element and return **null** if there is no such element. The **pollFirst()** and **pollLast()** methods remove and return the first and last element in the tree set, respectively.

TreeSet implements the **SortedSet** interface. To create a **TreeSet**, use a constructor, as shown in Figure 21.1. You can add objects into a *tree set* as long as they can be compared with each other.

tree set

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

Listing 21.4 gives an example of ordering elements using the **Comparable** interface. The preceding example in Listing 21.3 displays all the strings in their insertion order. This example rewrites the preceding example to display the strings in alphabetical order using the **TreeSet** class.

LISTING 21.4 TestTreeSet.java

```java
import java.util.*;

public class TestTreeSet {
  public static void main(String[] args) {
    // Create a hash set
    Set<String> set = new HashSet<>();

    // Add strings to the set
    set.add("London");
    set.add("Paris");
    set.add("New York");
    set.add("San Francisco");
    set.add("Beijing");
    set.add("New York");

    TreeSet<String> treeSet = new TreeSet<>(set);
    System.out.println("Sorted tree set: " + treeSet);

    // Use the methods in SortedSet interface
    System.out.println("first(): " + treeSet.first());
    System.out.println("last(): " + treeSet.last());
    System.out.println("headSet(\"New York\"): " +
      treeSet.headSet("New York"));
    System.out.println("tailSet(\"New York\"): " +
      treeSet.tailSet("New York"));

    // Use the methods in NavigableSet interface
    System.out.println("lower(\"P\"): " + treeSet.lower("P"));
    System.out.println("higher(\"P\"): " + treeSet.higher("P"));
    System.out.println("floor(\"P\"): " + treeSet.floor("P"));
    System.out.println("ceiling(\"P\"): " + treeSet.ceiling("P"));
    System.out.println("pollFirst(): " + treeSet.pollFirst());
    System.out.println("pollLast(): " + treeSet.pollLast());
    System.out.println("New tree set: " + treeSet);
  }
}
```

create hash set

create tree set

display elements

```
Sorted tree set: [Beijing, London, New York, Paris, San Francisco]
first(): Beijing
last(): San Francisco
headSet("New York"): [Beijing, London]
tailSet("New York"): [New York, Paris, San Francisco]
lower("P"): New York
higher("P"): Paris
floor("P"): New York
ceiling("P"): Paris
pollFirst(): Beijing
pollLast(): San Francisco
New tree set: [London, New York, Paris]
```

The example creates a hash set filled with strings, 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. Two elements `e1` and `e2` are considered duplicate for a `TreeSet` if `e1.compareTo(e2)` is `0` for `Comparable` and `e1.compare(e2)` is `0` for `Comparator`.

The elements in the set are sorted once you create a `TreeSet` object from a `HashSet` object using `new TreeSet<>(set)` (line 16). You may rewrite the program to create an instance of `TreeSet` using its no-arg constructor and add the strings into the `TreeSet` object.

`treeSet.first()` returns the first element in `treeSet` (line 20) and `treeSet.last()` returns the last element in `treeSet` (line 21). `treeSet.headSet("New York")` returns the elements in `treeSet` before New York (lines 22–23). `treeSet.tailSet("New York")` returns the elements in `treeSet` after New York, including New York (lines 24–25).

`treeSet.lower("P")` returns the largest element less than `P` in `treeSet` (line 28). `treeSet.higher("P")` returns the smallest element greater than `P` in `treeSet` (line 29). `treeSet.floor("P")` returns the largest element less than or equal to `P` in `treeSet` (line 30). `treeSet.ceiling("P")` returns the smallest element greater than or equal to `P` in `treeSet` (line 31). `treeSet.pollFirst()` removes the first element in `treeSet` and returns the removed element (line 32). `treeSet.pollLast()` removes the last element in `treeSet` and returns the removed element (line 33).

Note
All the concrete classes in Java Collections Framework (see Figure 20.1) have at least two constructors. One is the no-arg constructor that constructs an empty collection. The other constructs instances from a collection. Thus the `TreeSet` class has the constructor `TreeSet(Collection c)` for constructing a `TreeSet` from a collection `c`. In this example, `new TreeSet<>(set)` creates an instance of `TreeSet` from the collection `set`.

Tip
If you don't need to maintain a sorted set when updating a set, you should use a hash set because it takes less time to insert and remove elements in a hash set. When you need a sorted set, you can create a tree set from the hash set.

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

Listing 21.5 gives a program that demonstrates how to sort elements in a tree set using the `Comparator` interface.

LISTING 21.5 TestTreeSetWithComparator.java

```
 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 comparator
 6     Set<GeometricObject> set =
 7       new TreeSet<>(new GeometricObjectComparator());
 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 objects");
15     for (GeometricObject element: set)
16       System.out.println("area = " + element.getArea());
17   }
18 }
```

tree set

display elements

```
A sorted set of geometric objects
area = 4.0
area = 20.0
area = 5021.548245743669
```

The **GeometricObjectComparator** class is defined in Listing 20.4. The program creates a tree set of geometric objects using the **GeometricObjectComparator** for comparing the elements in the set (lines 6 and 7).

The **Circle** and **Rectangle** classes were defined in Section 13.2, Abstract Classes. They are all subclasses of **GeometricObject**. They are added to the set (lines 8–11).

Two circles of the same radius are added to the tree set (lines 9 and 10), but only one is stored because the two circles are equal (determined by the comparator in this case) and the set does not allow duplicates.

Check Point

21.2.1 How do you create an instance of **Set**? How do you insert a new element in a set? How do you remove an element from a set? How do you find the size of a set?

21.2.2 If two objects **o1** and **o2** are equal, what is **o1.equals(o2)** and **o1.hashCode() == o2.hashCode()**?

21.2.3 How do you traverse the elements in a set?

21.2.4 Suppose **set1** is a set that contains the strings **red**, **yellow**, and **green** and that **set2** is another set that contains the strings **red**, **yellow**, and **blue**. Answer the following questions:

- What are in **set1** and **set2** after executing **set1.addAll(set2)**?
- What are in **set1** and **set2** after executing **set1.add(set2)**?
- What are in **set1** and **set2** after executing **set1.removeAll(set2)**?
- What are in **set1** and **set2** after executing **set1.remove(set2)**?
- What are in **set1** and **set2** after executing **set1.retainAll(set2)**?
- What is in **set1** after executing **set1.clear()**?

21.2.5 Show the output of the following code:

```
import java.util.*;

public class Test {
  public static void main(String[] args) {
    LinkedHashSet<String> set1 = new LinkedHashSet<>();
    set1.add("New York");
    LinkedHashSet<String> set2 = set1;
    LinkedHashSet<String> set3 =
      (LinkedHashSet<String>)(set1.clone());
    set1.add("Atlanta");
    System.out.println("set1 is " + set1);
    System.out.println("set2 is " + set2);
    System.out.println("set3 is " + set3);
    set1.forEach(e -> System.out.print(e + " "));
  }
}
```

21.2.6 Show the output of the following code:

```
Set<String> set = new LinkedHashSet<>();
set.add("ABC");
set.add("ABD");
System.out.println(set);
```

21.2.7 What are the differences among `HashSet`, `LinkedHashSet`, and `TreeSet`?

21.2.8 How do you sort the elements in a set using the `compareTo` method in the `Comparable` interface? How do you sort the elements in a set using the `Comparator` interface? What would happen if you added an element that could not be compared with the existing elements in a tree set?

21.2.9 What will the output be if lines 6–7 in Listing 21.5 are replaced by the following code:

```
Set<GeometricObject> set = new HashSet<>();
```

21.2.10 Show the output of the following code:

```
Set<String> set = new TreeSet<>(
  Comparator.comparing(String::length));
set.add("ABC");
set.add("ABD");
System.out.println(set);
```

21.3 Comparing the Performance of Sets and Lists

Key Point

Sets are more efficient than lists for storing nonduplicate elements. Lists are useful for accessing elements through the index.

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

LISTING 21.6 SetListPerformanceTest.java

```java
import java.util.*;

public class SetListPerformanceTest {
  static final int N = 50000;

  public static void main(String[] args) {
    // Add numbers 0, 1, 2, ..., N - 1 to the array list
    List<Integer> list = new ArrayList<>();
    for (int i = 0; i < N; i++)
      list.add(i);
    Collections.shuffle(list); // Shuffle the array list

    // Create a hash set, and test its performance
    Collection<Integer> set1 = new HashSet<>(list);
    System.out.println("Member test time for hash set is " +
      getTestTime(set1) + " milliseconds");
    System.out.println("Remove element time for hash set is " +
      getRemoveTime(set1) + " milliseconds");

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

    // Create a tree set, and test its performance
    Collection<Integer> set3 = new TreeSet<>(list);
    System.out.println("Member test time for tree set is " +
      getTestTime(set3) + " milliseconds");
    System.out.println("Remove element time for tree set is " +
      getRemoveTime(set3) + " milliseconds");

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

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

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

    // Test if a number is in the collection
    for (int i = 0; i < N; i++)
      c.contains((int)(Math.random() * 2 * N));

    return System.currentTimeMillis() - startTime;
```

create test data
shuffle
a hash set
a linked hash set
a tree set
an array list
a linked list
start time
test membership
return execution time

```
57     }
58
59     public static long getRemoveTime(Collection<Integer> c) {
60       long startTime = System.currentTimeMillis();
61
62       for (int i = 0; i < N; i++)
63         c.remove(i);
64
65       return System.currentTimeMillis() - startTime;
66     }
67   }
```

remove from container

return execution time

```
Member test time for hash set is 20 milliseconds
Remove element time for hash set is 27 milliseconds
Member test time for linked hash set is 27 milliseconds
Remove element time for linked hash set is 26 milliseconds
Member test time for tree set is 47 milliseconds
Remove element time for tree set is 34 milliseconds
Member test time for array list is 39802 milliseconds
Remove element time for array list is 16196 milliseconds
Member test time for linked list is 52197 milliseconds
Remove element time for linked list is 14870 milliseconds
```

The program creates a list for numbers from **0** to **N-1** (for **N** = **50000**) (lines 8–10) and shuffles the list (line 11). From this list, the program creates a hash set (line 14), a linked hash set (line 21), a tree set (line 28), an array list (line 35), and a linked list (line 42). The program obtains the execution time for testing whether a number is in the hash set (line 16), linked hash set (line 23), tree set (line 30), array list (line 37), or linked list (line 44) and obtains the execution time for removing the elements from the hash set (line 18), linked hash set (line 25), tree set (line 32), array list (line 39), and linked list (line 46).

The **getTestTime** method invokes the **contains** method to test whether a number is in the container (line 54) and the **getRemoveTime** method invokes the **remove** method to remove an element from the container (line 63).

sets are better

As these runtimes illustrate, sets are much more efficient than lists for testing whether an element is in a set or a list. Therefore, the No-Fly list should be implemented using a hash set instead of a list, because it is much faster to test whether an element is in a hash set than in a list.

You may wonder why sets are more efficient than lists. The questions will be answered in Chapters 24 and 27 when we introduce the implementations of lists and sets.

Check Point

21.3.1 Suppose you need to write a program that stores unordered, nonduplicate elements, what data structure should you use?

21.3.2 Suppose you need to write a program that stores nonduplicate elements in the order of insertion, what data structure should you use?

21.3.3 Suppose you need to write a program that stores nonduplicate elements in increasing order of the element values, what data structure should you use?

21.3.4 Suppose you need to write a program that stores a fixed number of the elements (possibly duplicates), what data structure should you use?

21.3.5 Suppose you need to write a program that stores the elements in a list with frequent operations to append and delete elements at the end of the list, what data structure should you use?

21.3.6 Suppose you need to write a program that stores the elements in a list with frequent operations to insert and delete elements at the beginning of the list, what data structure should you use?

21.4 Case Study: Counting Keywords

Key Point

This section presents an application that counts the number of 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. Listing 21.7 gives this program.

LISTING 21.7 CountKeywords.java

```java
import java.util.*;
import java.io.*;

public class CountKeywords {
  public static void main(String[] args) throws Exception {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a Java source file: ");
    String filename = input.nextLine();

    File file = new File(filename);
    if (file.exists()) {
      System.out.println("The number of keywords in " + filename
        + " is " + countKeywords(file));
    }
    else {
      System.out.println("File " + filename + " does not exist");
    }
  }

  public static int countKeywords(File file) throws Exception {
    // Array of all Java keywords + true, false and null
    String[] keywordString = {"abstract", "assert", "boolean",
        "break", "byte", "case", "catch", "char", "class", "const",
        "continue", "default", "do", "double", "else", "enum",
        "extends", "for", "final", "finally", "float", "goto",
        "if", "implements", "import", "instanceof", "int",
        "interface", "long", "native", "new", "package", "private",
        "protected", "public", "return", "short", "static",
        "strictfp", "super", "switch", "synchronized", "this",
        "throw", "throws", "transient", "try", "void", "volatile",
        "while", "true", "false", "null"};

    Set<String> keywordSet =
      new HashSet<>(Arrays.asList(keywordString));
    int count = 0;

    Scanner input = new Scanner(file);

    while (input.hasNext()) {
      String word = input.next();
      if (keywordSet.contains(word))
        count++;
    }

    return count;
  }
}
```

enter a filename

file exists?

count keywords

keywords

keyword set

is a keyword?

```
Enter a Java source file: c:\ Welcome.java ↵Enter
The number of keywords in c:\ Welcome.java is 5
```

```
Enter a Java source file: c:\ TTT.java ↵Enter
File c:\ TTT.java does not exist
```

The program prompts the user to enter a Java source filename (line 7) and reads the filename (line 8). If the file exists, the **countKeywords** method is invoked to count the keywords in the file (line 13).

The **countKeywords** method creates an array of strings for the keywords (lines 22–31) and creates a hash set from this array (lines 33–34). It then reads each word from the file and tests if the word is in the set (line 41). If so, the program increases the count by **1** (line 42).

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.

Check Point

21.4.1 Will the **CountKeywords** program work if lines 33–34 are changed to

```
Set<String> keywordSet =
  new LinkedHashSet<>(Arrays.asList(keywordString));
```

21.4.2 Will the **CountKeywords** program work if lines 33–34 are changed to

```
List<String> keywordSet =
  new ArrayList<>(Arrays.asList(keywordString));
```

21.5 Maps

Key Point

You can create a map using one of its three concrete classes: **HashMap**, **LinkedHashMap**, *or* **TreeMap**.

map

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. A key and its corresponding value form an entry stored in a map, as shown in Figure 21.2a. Figure 21.2b shows a map in which each entry consists of a Social Security number as the key and a name as the value.

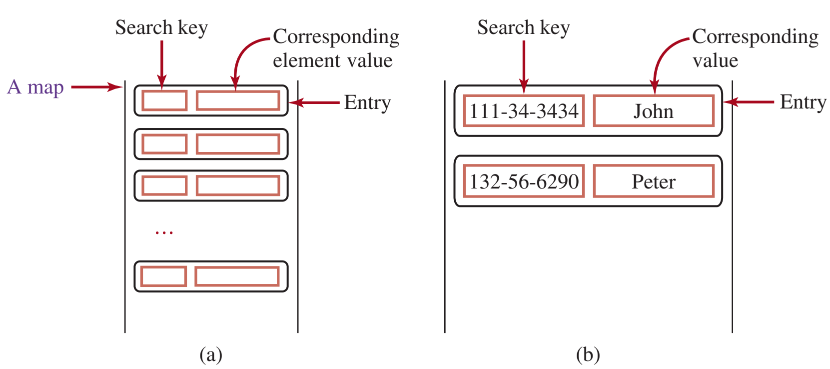

FIGURE 21.2 The entries consisting of key/value pairs are stored in a map.

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 21.3.

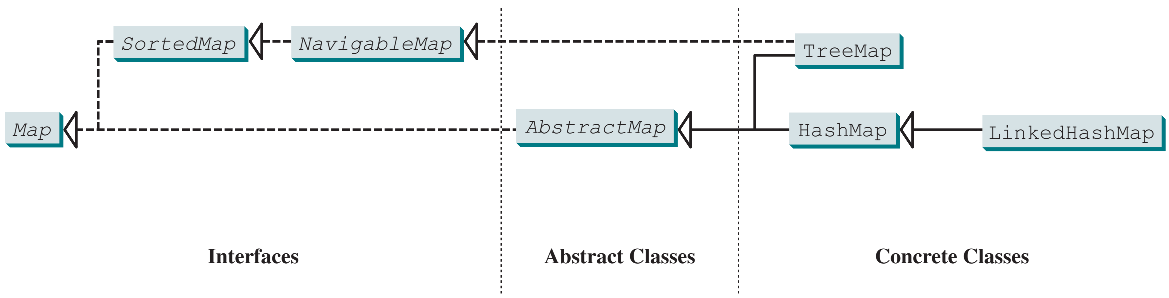

FIGURE 21.3 A map stores key/value pairs.

The **Map** interface provides the methods for querying, updating, and obtaining a collection of values and a set of keys, as shown in Figure 21.4.

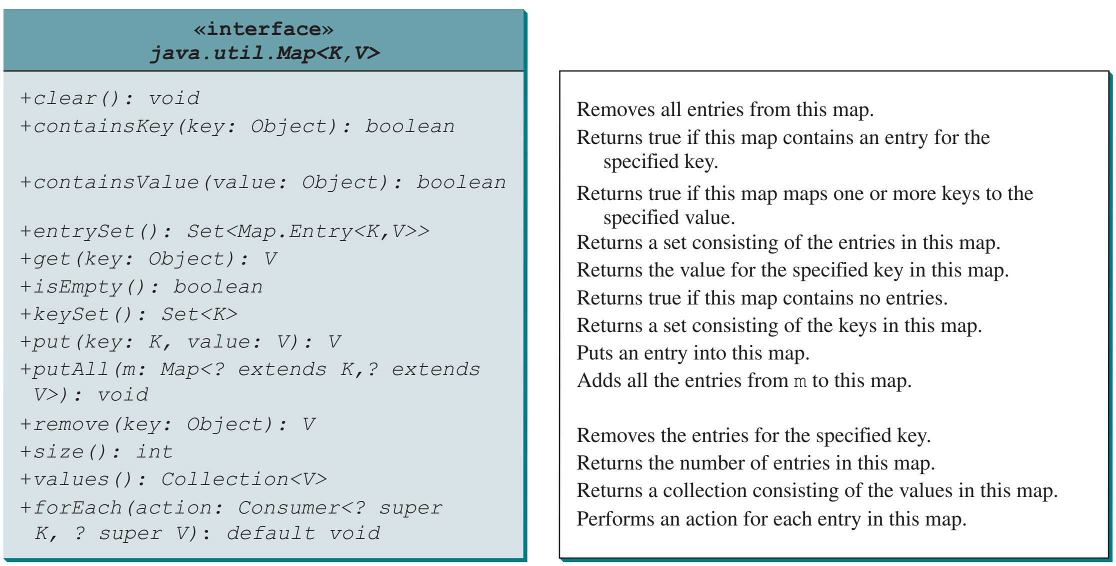

FIGURE 21.4 The **Map** interface maps keys to values.

The *update methods* include **clear**, **put**, **putAll**, and **remove**. The **clear()** method removes all entries from the map. The **put(K key, V value)** method adds an entry for the specified key and value in the map. If the map formerly contained an entry for this key, the old value is replaced by the new value, and the old value associated with the key is returned. The **putAll(Map m)** method adds all entries in **m** to this map. The **remove(Object key)** method removes the entry for the specified key from the map. update methods

The *query methods* include **containsKey**, **containsValue**, **isEmpty**, and **size**. The **containsKey(Object key)** method checks whether the map contains an entry for the specified key. The **containsValue(Object value)** method checks whether the map contains an entry for this value. The **isEmpty()** method checks whether the map contains any entries. The **size()** method returns the number of entries in the map. query methods

keySet()
values()
entrySet()

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, as shown in Figure 21.5. Each entry in the set is a key/value pair in the underlying map.

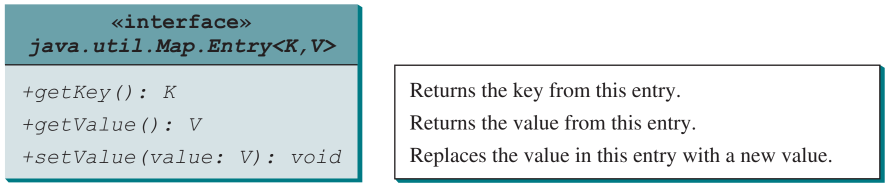

FIGURE 21.5 The `Map.Entry` interface operates on an entry in the map.

forEach method

Java 8 added a default `forEach` method in the `Map` interface for performing an action on each entry in the map. This method can be used like an iterator for traversing the entries in the map.

AbstractMap

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

concrete implementation

The `HashMap`, `LinkedHashMap`, and `TreeMap` classes are three *concrete implementations* of the `Map` interface, as shown in Figure 21.6.

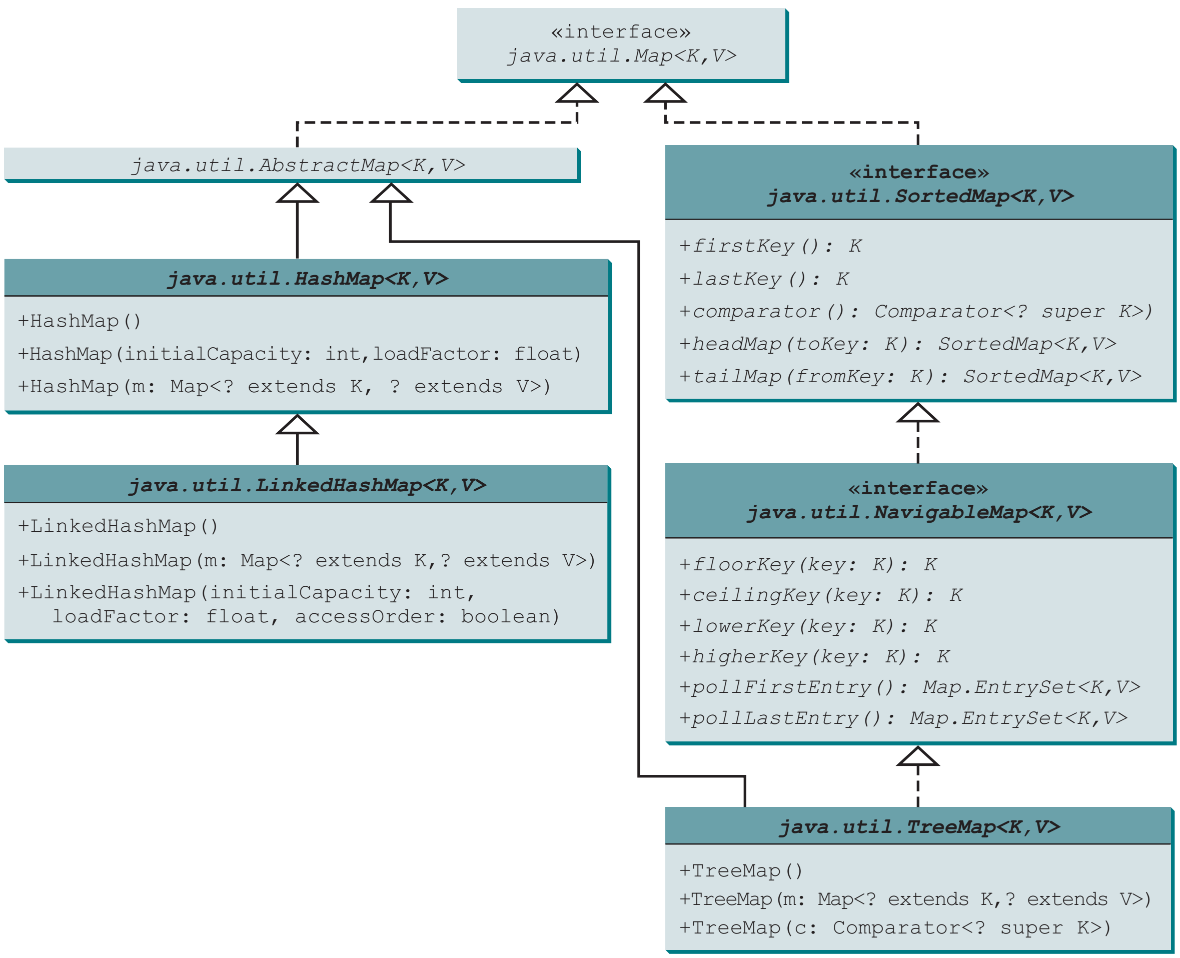

FIGURE 21.6 The Java Collections Framework provides three concrete map classes.

HashMap
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 in the map. 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)**.

insertion order
access order

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 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

SortedMap is a subinterface of **Map**, which guarantees the entries in the map are sorted. In addition, it provides the methods **firstKey()** and **lastKey()** for returning the first and the 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

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 the last entry in the tree map, respectively.

Hashtable

Note

Prior to Java 2, **java.util.Hashtable** was used for mapping keys with values. **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 the update methods in **Hashtable** are synchronized.

hash map
linked hash map
tree map

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.

Listing 21.8 TestMap.java

```
import java.util.*;

public class TestMap {
  public static void main(String[] args) {
    // Create a HashMap
    Map<String, Integer> hashMap = new HashMap<>();
    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 preceding HashMap
    Map<String, Integer> treeMap = new TreeMap<>(hashMap);
    System.out.println("Display entries in ascending order of key");
```

create map
add entry
tree map

linked hash map

forEach method

```
18      System.out.println(treeMap);
19
20      // Create a LinkedHashMap
21      Map<String, Integer> linkedHashMap =
22        new LinkedHashMap<>(16, 0.75f, true);
23      linkedHashMap.put("Smith", 30);
24      linkedHashMap.put("Anderson", 31);
25      linkedHashMap.put("Lewis", 29);
26      linkedHashMap.put("Cook", 29);
27
28      // Display the age for Lewis
29      System.out.println("\nThe age for " + "Lewis is " +
30        linkedHashMap.get("Lewis"));
31
32      System.out.println("Display entries in LinkedHashMap");
33      System.out.println(linkedHashMap);
34
35      // Display each entry with name and age
36      System.out.print("\nNames and ages are ");
37      treeMap.forEach(
38        (name, age) -> System.out.print(name + ": " + age + " "));
39    }
40  }
```

```
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}


Names and ages are Anderson: 31 Cook: 29 Lewis: 29 Smith: 30
```

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

All the concrete classes that implement the **Map** interface have at least two constructors. One is the no-arg constructor that constructs an empty map, and the other constructs a map from an instance of **Map**. Thus, **new TreeMap<>(hashMap)** (line 16) constructs a tree map from a hash map.

You can create an insertion- or access-ordered linked hash map. An access-ordered linked hash map is created in lines 21–22. The most recently accessed entry is placed at the end of the map. The entry with the key **Lewis** is last accessed in line 30, so it is displayed last in line 33.

It is convenient to process all the entries in the map using the **forEach** method. The program uses a **forEach** method to display a name and its age (lines 37–38).

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**.

Check Point

21.5.1 How do you create an instance of **Map**? How do you add an entry to a map consisting of a key and a value? How do you remove an entry from a map? How do you find the size of a map? How do you traverse entries in a map?

21.5.2 Describe and compare `HashMap`, `LinkedHashMap`, and `TreeMap`.

21.5.3 Show the output of the following code:

```
import java.util.*;
public class Test {
  public static void main(String[] args) {
    Map<String, String> map = new LinkedHashMap<>();
    map.put("123", "John Smith");
    map.put("111", "George Smith");
    map.put("123", "Steve Yao");
    map.put("222", "Steve Yao");
    System.out.println("(1) " + map);
    System.out.println("(2) " + new TreeMap<String, String>(map));
    map.forEach((k, v) -> {
      if (k.equals("123")) System.out.println(v);});
  }
}
```

21.6 Case Study: Occurrences of Words

Key Point

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 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; for example, `Good` is treated the same as `good`.

Listing 21.9 gives the solution to the problem.

LISTING 21.9 CountOccurrenceOfWords.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
    Map<String, Integer> map = new TreeMap<>();

    String[] words = text.split("[\\s+\\p{P}]");
    for (int i = 0; i < words.length; i++) {
      String key = words[i].toLowerCase();

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

    // Display key and value for each entry
```

tree map

split string

add entry

update entry

display entry

```
29      map.forEach((k, v) -> System.out.println(k + "\t" + v));
30    }
31  }
```

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

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

The program extracts a word from a text using the **split** method (line 12) in the **String** class (see Section 10.10.4 and Appendix H). The text is split into words using a whitespace **\s** or punctuation **\p{P}** as a delimiter. For each word extracted, the program checks whether it is already stored as a key in the map (line 17). If not, a new pair consisting of the word and its initial count (**1**) is stored in the map (line 18). Otherwise, the count for the word is incremented by **1** (lines 21–23).

The program displays the count and the key in each entry using the **forEach** method in the **Map** class (line 29).

Since the map is a tree map, the entries are displayed in increasing order of words. To display them in ascending order of the occurrence counts, see Programming Exercise 21.8.

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

Java Collections Framework provides comprehensive support of organizing and manipulating data. Suppose you wish to display the words in increasing order of their occurrence values, how do you modify the program? This can be done simply by creating a list of map entries and creating a **Comparator** for sorting the entries on their values as follows:

```
List<Map.Entry<String, Integer>> entries =
  new ArrayList<>(map.entrySet());
Collections.sort(entries, (entry1, entry2) -> {
  return entry1.getValue().compareTo(entry2.getValue()); });
for (Map.Entry<String, Integer> entry: entries) {
  System.out.println(entry.getKey() + "\t" + entry.getValue());
}
```

Check Point

21.6.1 Will the **CountOccurrenceOfWords** program work if line 10 is changed to

```
Map<String, int> map = new TreeMap<>();
```

21.6.2 Will the **CountOccurrenceOfWords** program work if line 17 is changed to

```
if (map.get(key) == null) {
```

21.6.3 Will the **CountOccurrenceOfWords** program work if line 29 is changed to

```
for (String key: map)
  System.out.println(key + "\t" + map.getValue(key));
```

21.6.4 How do you simplify the code in lines 17–24 in Listing 21.9 in one line using a conditional expression?

21.7 Singleton and Unmodifiable Collections and Maps

You can create singleton sets, lists, and maps and unmodifiable sets, lists, and maps using the static methods in the `Collections` *class.*

Key Point

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.

FIGURE 21.7 The `Collections` class contains the static methods for creating singleton and read-only sets, lists, and maps.

The `Collections` class defines three constants—`EMPTY_SET`, `EMPTY_LIST`, and `EMPTY_MAP`—for an empty set, an empty list, and an 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.

read-only view

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(SortedSet s)`. This type of view is like a reference to the actual collection. However, you cannot modify the collection through a read-only view. Attempting to modify a collection through a read-only view will cause an `UnsupportedOperationException`.

In JDK 9, you can use the static `Set.of(e1, e2, ...)` method to create an immutable set and `Map.of(key1, value1, key2, value2, ...)` method to create an immutable map.

Check Point

21.7.1 What is wrong in the following code?

```
Set<String> set = Collections.singleton("Chicago");
set.add("Dallas");
```

21.7.2 What happens when you run the following code?

```
List list = Collections.unmodifiableList(Arrays.asList("Chicago",
  "Boston"));
list.remove("Dallas");
```

Key Terms

hash map 831
hash set 816
linked hash map 831
linked hash set 820
map 828
read-only view 835
set 816
tree map 831
tree set 821

Chapter Summary

1. A set stores nonduplicate elements. To allow duplicate elements to be stored in a collection, you need to use a list.

2. A *map* stores key/value pairs. It provides a quick lookup for a value using a key.

3. Three types of sets are supported: `HashSet`, `LinkedHashSet`, and `TreeSet`. `HashSet` stores elements in an unpredictable order. `LinkedHashSet` stores elements in the order they were inserted. `TreeSet` stores elements sorted. `HashSet`, `LinkedHashSet`, and `TreeSet` are subtypes of `Collection`.

4. The `Map` interface maps keys to the elements. 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 can map to at most one value. The `Map` interface provides the methods for querying, updating, and obtaining a collection of values and a set of keys.

5. Three types of maps are supported: `HashMap`, `LinkedHashMap`, and `TreeMap`. `HashMap` is efficient for locating a value, inserting an entry, and deleting an entry. `LinkedHashMap` supports ordering of the entries in the map. 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 accessed to most recently (*access order*). `TreeMap` is efficient for traversing the keys in a sorted order. The keys can be sorted using the `Comparable` interface or the `Comparator` interface.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

Programming Exercises

Sections 21.2–21.4

21.1 (*Perform set operations on hash sets*) Create two linked hash sets {`"George"`, `"Jim"`, `"John"`, `"Blake"`, `"Kevin"`, `"Michael"`} and {`"George"`, `"Katie"`, `"Kevin"`, `"Michelle"`, `"Ryan"`} and find their union, difference, and intersection. (You can clone the sets to preserve the original sets from being changed by these set methods.)

21.2 (*Display nonduplicate words in ascending order*) Write a program that reads words from a text file and displays all the nonduplicate words in ascending order. The text file is passed as a command-line argument.

****21.3** (*Count the keywords in Java source code*) Revise the program in Listing 21.7. If a keyword is in a comment or in a string, don't count it. Pass the Java file name from the command line. Assume the Java source code is correct and line comments and paragraph comments do not overlap.

***21.4** (*Count consonants and vowels*) Write a program that prompts the user to enter a text file name and displays the number of vowels and consonants in the file. Use a set to store the vowels **A**, **E**, **I**, **O**, and **U**.

*****21.5** (*Syntax highlighting*) Write a program that converts a Java file into an HTML file. In the HTML file, the keywords, comments, and literals are displayed in bold navy, green, and blue, respectively. Use the command line to pass a Java file and an HTML file. For example, the following command

```
java Exercise21_05 Welcome.java Welcome.html
```

converts **Welcome.java** into **Welcome.html**. Figure 21.8a shows a Java file. The corresponding HTML file is shown in Figure 21.8b.

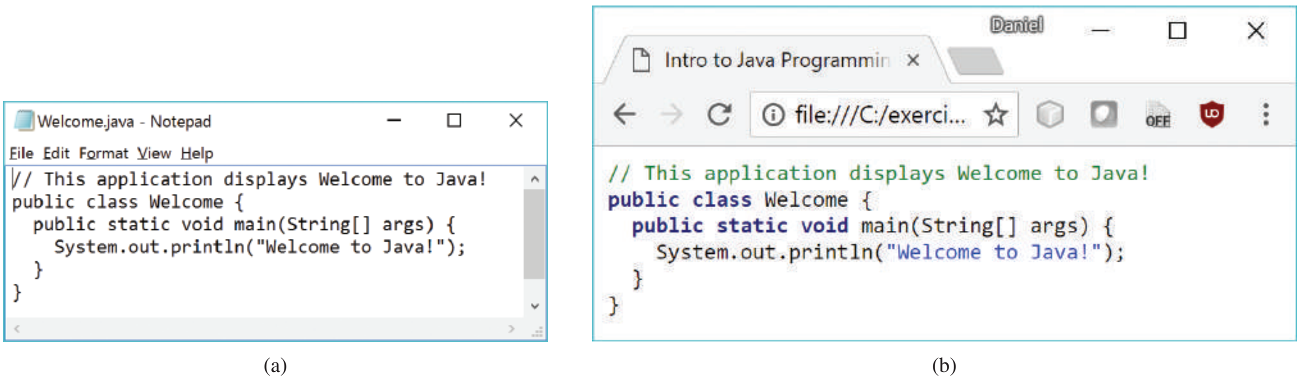

FIGURE 21.8 The Java code in plain text in (a) is displayed in HTML with syntax highlighted in (b). *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Sections 21.5–21.7

***21.6** (*Count the occurrences of numbers entered*) Write a program that reads an unspecified number of integers and finds the one that has the most occurrences. The input ends when the input is **0**. For example, if you entered **2 3 40 3 5 4 –3 3 3 2 0**, the number **3** occurred most often. If not one but several numbers have the most occurrences, all of them should be reported. For example, since **9** and **3** appear twice in the list **9 30 3 9 3 2 4**, both occurrences should be reported.

****21.7** (*Revise Listing 21.9, CountOccurrenceOfWords.java*) Rewrite Listing 21.9 to display the words in ascending order of occurrence counts.

****21.8** (*Count the occurrences of words in a text file*) Rewrite Listing 21.9 to read the text from a text file. The text file is passed as a command-line argument. Words are delimited by whitespace characters, punctuation marks (**, ; . : ?**), quotation marks (**' "**), and parentheses. Count words in case-insensitive fashion (e.g., consider **Good** and **good** to be the same word). The words must start with a letter. Display the output in alphabetical order of words, with each word preceded by its occurrence count.

****21.9** (*Guess the capitals using maps*) Rewrite Programming Exercise 8.37 to store pairs of each state and its capital in a map. Your program should prompt the user to enter a state and should display the capital for the state.

***21.10** (*Count the occurrences of each keyword*) Rewrite Listing 21.7, CountKeywords.java to read in a Java source-code file and count the occurrence of each keyword in the file, but don't count the keyword if it is in a comment or in a string literal.

****21.11** (*Baby name popularity ranking*) Use the data files from Programming Exercise 12.31 to write a program that enables the user to select a year, gender, and enter a name to display the ranking of the name for the selected year and gender, as shown in Figure 21.9. To achieve the best efficiency, create two arrays for boy's names and girl's names, respectively. Each array has 10 elements for 10 years. Each element is a map that stores a name and its ranking in a pair with the name as the key.

Figure 21.9 The user selects a year and gender, enters a year, and clicks the Find Ranking button to display the ranking. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****21.12** (*Name for both genders*) Write a program that prompts the user to enter one of the filenames described in Programming Exercise 12.31 and displays the names that are used for both genders in the file. Use sets to store names and find common names in two sets. Here is a sample run:

```
Enter a file name for baby name ranking: babynamesranking2001.txt ↵Enter
69 names used for both genders
They are Tyler Ryan Christian ...
```

****21.13** (*Baby name popularity ranking*) Revise Programming Exercise 21.11 to prompt the user to enter year, gender, and name and display the ranking for the name. Prompt the user to enter another inquiry or exit the program. Here is a sample run:

```
Enter the year: 2010 ↵Enter
Enter the gender: M ↵Enter
Enter the name: Javier ↵Enter
Boy name Javier is ranked #190 in year 2010
Enter another inquiry? Y ↵Enter
Enter the year: 2001 ↵Enter
Enter the gender: F ↵Enter
Enter the name: Emily ↵Enter
Girl name Emily is ranked #1 in year 2001
Enter another inquiry? N ↵Enter
```

****21.14** (*Web crawler*) Rewrite Listing 12.18, WebCrawler.java, to improve the performance by using appropriate new data structures for `listOfPendingURLs` and `listofTraversedURLs`.

****21.15** (*Addition quiz*) Rewrite Programming Exercise 11.16 to store the answers in a set rather than a list.