16.10 Maps

`Map`s associate *keys* to *values*. The keys in a `Map` must be *unique*, but the associated values need not be. If a `Map` contains both unique keys and unique values, it's said to implement a **one-to-one mapping**. If only the keys are unique, the `Map` is said to implement a **many-to-one mapping**—many keys can map to one value.

`Map`s differ from `Set`s in that `Map`s contain keys and values, whereas `Set`s contain only values. Two classes that implement interface `Map` are `HashMap` and `TreeMap`. `HashMap`s store elements in hash tables, and `TreeMap`s store elements in trees. This section discusses hash tables and provides an example that uses a `HashMap` to store key–value pairs. **Interface** `SortedMap` extends `Map` and maintains its keys in *sorted* order—either the elements' *natural* order or an order specified by a `Comparator`. Class `TreeMap` implements `SortedMap`.

Map Implementation with Hash Tables

When a program creates objects, it may need to store and retrieve them efficiently. Storing and retrieving information with arrays is efficient if some aspect of your data directly matches a numerical key value and if the *keys are unique* and tightly packed. If you have 100 employees with nine-digit social security numbers and you want to store and retrieve employee data by using the social security number as a key, the task will require an array with over 800 million elements, because nine-digit Social Security numbers must begin with 001–899 (excluding 666) as per the Social Security Administration's website

```
http://www.socialsecurity.gov/employer/randomization.
```

This is impractical for virtually all applications that use social security numbers as keys. A program having so large an array could achieve high performance for both storing and retrieving employee records by simply using the social security number as the array index.

Numerous applications have this problem—namely, that either the keys are of the wrong type (e.g., not positive integers that correspond to array subscripts) or they're of the right type, but *sparsely* spread over a *huge range*. What is needed is a high-speed scheme for converting keys such as social security numbers, inventory part numbers and the like into unique array indices. Then, when an application needs to store something, the scheme can convert the application's key rapidly into an index, and the record can be stored at that slot in the array. Retrieval is accomplished the same way: Once the application has a key for which it wants to retrieve a data record, the application simply applies the conversion to the key—this produces the array index where the data is stored and retrieved.

The scheme we describe here is the basis of a technique called **hashing**. Why the name? When we convert a key into an array index, we literally scramble the bits, forming a kind of “mishmashed,” or hashed, number. The number actually has no real significance beyond its usefulness in storing and retrieving a particular data record.

A glitch in the scheme is called a **collision**—this occurs when two different keys “hash into” the same cell (or element) in the array. We cannot store two values in the same space, so we need to find an alternative home for all values beyond the first that hash to a particular array index. There are many schemes for doing this. One is to “hash again” (i.e., to apply another hashing transformation to the key to provide the next candidate cell in the array). The hashing process is designed to *distribute* the values throughout the table, so the assumption is that an available cell will be found with just a few hashes.

Another scheme uses one hash to locate the first candidate cell. If that cell is occupied, successive cells are searched in order until an available cell is found. Retrieval works the same way: The key is hashed once to determine the initial location and check whether it contains the desired data. If it does, the search is finished. If it does not, successive cells are searched linearly until the desired data is found.

The most popular solution to hash-table collisions is to have each cell of the table be a hash “bucket,” typically a linked list of all the key–value pairs that hash to that cell. This is the solution that Java’s `HashMap` class (from package `java.util`) uses. `HashMap` implements the `Map` interface.

A hash table’s **load factor** affects the performance of hashing schemes. The load factor is the ratio of the number of occupied cells in the hash table to the total number of cells in the hash table. The closer this ratio gets to 1.0, the greater the chance of collisions.

Performance Tip 16.1

The load factor in a hash table is a classic example of a memory-space/execution-time trade-off: By increasing the load factor, we get better memory utilization, but the program runs slower, due to increased hashing collisions. By decreasing the load factor, we get better program speed, because of reduced hashing collisions, but we get poorer memory utilization, because a larger portion of the hash table remains empty.

Computer-science students study hashing schemes in courses called “Data Structures” and “Algorithms.” Class `HashMap` enables you to use hashing without having to implement hash-table mechanisms—a classic example of reuse. This concept is profoundly important in our study of object-oriented programming. As discussed in earlier chapters, classes encapsulate and hide complexity (i.e., implementation details) and offer user-friendly interfaces. Properly crafting classes to exhibit such behavior is one of the most valued skills in the field of object-oriented programming. Figure 16.17 uses a `HashMap` to count the number of occurrences of each word in a string.

Line 12 creates an empty `HashMap` with a *default initial capacity* (16 elements) and a default load factor (0.75)—these defaults are built into the implementation of `HashMap`. When the number of occupied slots in the `HashMap` becomes greater than the capacity times the load factor, the capacity is doubled automatically. `HashMap` is a generic class that takes two type arguments—the type of key (i.e., `String`) and the type of value (i.e., `Integer`). Recall that the type arguments passed to a generic class must be reference types, hence the second type argument is `Integer`, not `int`.

```
1   // Fig. 16.17: WordTypeCount.java
2   // Program counts the number of occurrences of e
3   import java.util.Map;
4   import java.util.HashMap;
5   import java.util.Set;
6   import java.util.TreeSet;
7   import java.util.Scanner;
8
9   public class WordTypeCount {
10     public static void main(String[] args) {
11        // create HashMap to store String keys and
12        Map<String, Integer= myMap = new HashMap<>
13
14        createMap(myMap); // create map based on u
15        displayMap(myMap); // display map content
16     }
17
18     // create map from user input
```

```java
   private static void createMap(Map<String, Int
      Scanner scanner = new Scanner(System.in);
      System.out.println("Enter a string:"); //
      String input = scanner.nextLine();

      // tokenize the input
      String[] tokens = input.split(" ");

      // processing input text
      for (String token : tokens) {
         String word = token.toLowerCase(); // g

         // if the map contains the word
         if (map.containsKey(word)) { // is word
            int count = map.get(word); // get cu
            map.put(word, count + 1); // increme
         }
         else {
            map.put(word, 1); // add new word wi
         }
      }
   }

   // display map content
   private static void displayMap(Map<String, In
      Set<String> keys = map.keySet(); // get ke

      // sort keys
      TreeSet<String> sortedKeys = new TreeSet<>

      System.out.printf("%nMap contains:%nKey\t\

      // generate output for each key in map
      for (String key : sortedKeys) {
         System.out.printf("%-10s%10s%n", key, m
      }

      System.out.printf(
         "%nsize: %d%nisEmpty: %b%n", map.size()
   }
```

```
}
```

```
Enter a string:
this is a sample sentence with several words this is
sentence with several different words

Map contains:
```

Key	Value
a	1
another	1
different	1
is	2
sample	2
sentence	2
several	2
this	2
with	2
words	2

```
size: 10
isEmpty: false
```

Fig. 16.17

Program counts the number of occurrences of each word in a

`String`.

Line 14 calls method `createMap` (lines 19–40), which uses a `Map` to store the number of occurrences of each word in the sentence. Line 22 obtains the user input, and line 25 tokenizes it. For each token, lines 28–39 convert the token to lowercase letters (line 29), then call `Map` **method** `containsKey` (line 32) to determine whether the word is in the map (and thus has occurred previously in the string). If the `Map` does *not* contain the word, line 37 uses `Map` **method** `put` to create a new entry, with the word as the key and an `Integer` object containing `1` as the value. Autoboxing occurs when the program passes integer `1` to method `put`, because the map stores the number of occurrences as an `Integer`. If the word does exist in the map, line 33 uses `Map` **method** `get` to obtain the key's associated value (the count) in the map. Line 34 increments that value and uses `put` to replace the key's associated value. Method `put` returns the key's prior associated value, or `null` if the key was not in the map.

Error-Prevention Tip 16.1

Always use immutable keys with a `Map`*. The key determines where the corresponding value is placed. If the key has changed since the insert operation, when you subsequently attempt to retrieve that value, it might not be found. In this chapter's examples, we use* `String`*s as keys and* `String`*s* *are immutable.*

Method `displayMap` (lines 43–58) displays all the entries in the map. It uses `HashMap` **method** `keySet` (line 44) to get a set of the keys. The keys have type `String` in the `map`, so method `keySet` returns a generic type `Set` with type parameter specified to be `String`. Line 47 creates a `TreeSet` of the keys, in which the keys are sorted. Lines 52–54 access each key and its value in the map. Line 53 displays each key and its value using format specifier `%-10s` to *left align* each key and format specifier `%10s` to *right align* each value. The keys are displayed in *ascending* order. Line 57 calls `Map` **method** `size` to get the number of key–value pairs in the `Map`, and calls `Map` **method** `isEmpty`, which returns a `boolean` indicating whether the `Map` is empty.