32 Sets and Maps

1 Sets

Java provides a suite of classes for managing collections of data. They exist in a library called the *collections framework*. The classes in the collections framework are called the *collection classes*. Java supports three categories of collection: *sets*, *lists* (also called *sequences*), and *maps*. For each type of collection, there is a choice of two classes that behave similarly but differ in their implementation. The six classes are as follows:

Sets	*Lists*	*Maps*
`HashSet<T>`	`ArrayList<T>`	`HashMap<T,U>`
`TreeSet<T>`	`LinkedList<T>`	`TreeMap<T,U>`

In this chapter we deal with sets and maps. Throughout, we will use `T` as the type parameter, or `T` and `U` where there are two type parameters. The classes in the collections framework are in package `java.util` which must be explicitly imported.

A *set* is a collection of items without repetitions, and without any ordering on the items. By "without repetitions" is meant that an item either belongs to a set or it does not — it cannot occur twice or three times. It is not an error to "add" an item to a set when it is already a member – the second addition simply has no effect. When the items of a collection are stored in an array `c`, say, we think of them as being stored "in a row" and so we can refer to the first item (`c[0]`), the second item (`c[1]`), and so on. In the case of sets, however, we have to envisage the collection more like a fistful of items, and so we can't sensibly talk about the first or second item and so on. The technical way of expressing this is to say that the items are not ordered. An example of a set is the collection of standard mathematical symbols (such as +, /, <, ≠,√ etc.). Another example is the collection of the names of all the countries in Europe. This is a set because a given country is either present or absent, and there is no implied order on the names. Of course, we may think of names of countries as being ordered alphabetically, or by population from smaller to larger, but they are not ordered within the set. When we write down the items in a set we necessarily have to write them in some order, but the order in which we write then is just accidental or convenient: any order is as good as any other. An example of a

collection that is not a set is the collection of coins in your pocket. It is not a set because there is every possibility that you have more than one 20¢ coin, say. Even if your pocket has just one 20¢ coin, adding another one will leave you with two 20¢ coins. The items in a collection are said to be *elements* or *members* of the collection.

All the collection classes handle collections of objects only. Wrapper classes must be employed when we insert integers, reals, booleans, or other primitive types, although we can rely on automatic boxing and unboxing to eliminate the drudgery.

Java provides two classes that support sets: **`HashSet<T>`** and **`TreeSet<T>`**. They have the same methods but different implementations, and so a choice of one over the other will be based on performance criteria with respect to the application we have in mind. The constructors for the set classes are as follows:

HashSet()	HashSet(*Collection<T>*)
TreeSet()	TreeSet(*Collection<T>*)

`new HashSet()` and **`new TreeSet()`** each creates an empty set. **`new HashSet(c)`** and **`new TreeSet(c)`** each creates a set whose elements are initially the same as those in collection `c`. We write `Collection` to stand for any one of `HashSet` `TreeSet` `LinkedList`, or `ArrayList`. For example, `HashSet(Collection<T>)` is shorthand for the four constructors `HashSet(HashSet<T>)`, `HashSet(TreeSet<T>)`, `HashSet(LinkedList<T>)`, and `HashSet(ArrayList<T>)`. (Remark for readers who have studied inheritance: whenever a collection of elements of type `T` is expected, it is allowable to supply a collection whose elements inherit from `T`.)

There are four basic operations on sets: add an item, remove an item, determine whether a given item occurs in the set, and determine the size of the set (i.e. the number of elements it contains). `HashSet<T>` and `TreeSet<T>` provide the following methods for the basic operations:

```
boolean add(T)
boolean remove(T)
boolean contains(T)
int size()
```

Java	*Using set theory*
`s.add(o)`	`s=s∪{o}`
`s.remove(o)`	`s=s\{o}`
`s.contains(o)`	`o∈s`
`s.size()`	`#s`

`s.add(o)` adds `o` to set `s`. `s` is unchanged if it already contains `o`. A boolean is returned indicating whether `s` was changed (the returned boolean is often ignored). **`s.remove(o)`** removes `o` from set `s` if it is present. `s` is unchanged if it does not contain `o`. A boolean is returned indicating whether the set was changed (the returned boolean is often ignored). **`s.contains(o)`** returns a boolean indicating whether `o` is an element of set `s`. **`s.size()`** returns the number of elements in set `s` (the size of a set is also called its *cardinality*). If you happen to be familiar with set theory from a

mathematics course, it may help to understand the above methods in terms of standard set operations. They are summarised in the accompanying table, where ∪ is the set union symbol, \ is the set difference symbol, ∈ is the set membership symbol, and # is the set cardinality symbol. Remember that these symbols cannot be used in Java programs. If you haven't previously studied sets, you may safely ignore the table.

Both `HashSet` and `TreeSet` also provide

```
String toString()
```

which returns a string representation of the set (encased in square brackets). Each element is represented by the string returned by its `toString()` method. In the case of `TreeSet`, the elements are listed in ascending order. Even though `toString()` generates the elements in ascending order, it is not the case that they exist in the set in ascending order. The elements of a set do not have any implied ordering at all.

Changing the state of an object while it is a member of a set can lead to strange behaviour and should be avoided.

The following code illustrates the behaviour of the above constructors and methods.

```
TreeSet<String> s =
        new TreeSet<String>();          // s is empty
s.add("dog");                           // s: dog
s.add("cat");                           // s: dog, cat
s.add("bird");                          // s: dog, cat, bird
s.add("mouse");                         // s: dog, cat, bird, mouse
s.add("dog");                           // s: dog, cat, bird, mouse
HashSet<String> h
        = new HashSet<String>(s);       // h: dog, cat, bird, mouse
s.remove("bird");                       // s: dog, cat, mouse
boolean b = s.remove("rat");            // s: dog, cat, mouse
System.out.print(b);                    // false appears
System.out.print(s);                    // [cat, dog, mouse] appears
System.out.print(s.contains("cat"));    // True appears
System.out.print(s.size());             // 3 appears
System.out.print(h);                    // [cat, mouse, dog, bird] appears
```

Note that the members of set `s` (whose type is `TreeSet`) are displayed in ascending order, but not those of `h` (because its type is `HashSet`) .

Example 1: counting distinct words

The following program reads a sequence of words from the keyboard, and displays the number of distinct words in the input, followed by a list of the words in alphabetical order. For example, for the input

```
a horse a horse my kingdom for a horse
```

the output is

```
There were 5 unique words, as follows:
[a, for, horse, kingdom, my]
```

(the square brackets are just a by-product of the use of the set classes).

```
import java.util.*;
class Uniques {
   public static void main(String[] args) {
      TreeSet<String> ws = new TreeSet<String> (); // TreeSet for alphabetic order
       while (!Console.endOfFile()) {
          String w = Console.readToken();
          ws.add(w);
       }
       System.out.println("There were " + ws.size() + " unique words, as follows:");
       System.out.print(ws);
   }
}
```

If you replace both occurrences of `TreeSet` with `HashSet`, the program will generate the same output with the minor difference that the words in the output may not be in alphabetic order.

`TreeSet` carries out the basic operations extremely quickly, while `HashSet` employs a technique called *hashing* which, remarkably, allows them to be completed almost instantaneously. We can express this more technically for readers who have studied the chapter on running times: `TreeSet` carries out the basic operations in logarithmic time (actually size determination can be done in constant time, but this is not guaranteed), and `HashSet` carries them out in constant time. In short, if your program employs a set on which only the basic operations are carried out, it is best to implement it as a set of type `HashSet`.

2 Sets: algebraic operations

The following methods of `HashSet<T>` and `TreeSet<T>` implement what are known as *algebraic* operations on sets:

```
void clear()
boolean isEmpty()
boolean addAll(Collection<T>)
boolean retainAll(Collection<?>)
boolean removeAll(Collection<?>)
boolean containsAll(Collection<?>)
```

Java	*Using set theory*
`s.clear()`	s=∅
`s.isEmpty()`	s==∅
`s.addAll(t)`	s=s∪t
`s.retainAll(t)`	s=s∩t
`s.removeAll(t)`	s=s\t
`s.containsAll(t)`	t⊆s

We use `Collection` as before. By writing `addAll(Collection<T>)`, for example, we imply that if `h1` and `h2` are instances of `HashSet<T>`, and `t1` and `t2` are instances of `TreeSet<T>`, then all of `t1.addAll(t2)`, `t1.addAll(h1)`, `h1.addAll(h2)`, and `h1.addAll(t1)` are allowed (among others). We write <?> to stand for an arbitrary type that may or may not differ from `T`. For example, if `t1` is an instance of `TreeSet<String>` and `t2` is an instance of `TreeSet<Integer>`, then `t1.retainAll(t2)` is actually legal although we will have no use for such generality.

`s.clear()` removes all elements from set `s`. **`s.isEmpty()`** returns a boolean indicating whether set `s` is empty (i.e. contains no elements). **`s.addAll(t)`** adds the elements in `t` to set `s` (without changing `t`). If any elements of `t` already occur in `s`, their addition has no effect – remember that an element cannot occur twice in a set. `s.addAll(t)` returns a boolean indicating whether `s` changed as a result of the call (the returned boolean is often ignored). **`s.retainAll(t)`** removes from set `s` those elements that are not in set `t`. A boolean is returned indicating whether `s` changed as a result of the call (the returned boolean is often ignored). **`s.removeAll(t)`** removes from set `s` those elements that are also members of set `t` (without changing `t`). A boolean is returned indicating whether `s` changed (the returned boolean is often ignored). **`s.containsAll(t)`** returns a boolean indicating whether set `s` contains all the elements of set `t` (it may or may not contain elements other than those in `t`). The relationship between the above methods and certain operations in set theory are summarised in the accompanying table, where ∅ is the symbol representing the empty set, ∩ is the symbol for set intersection, and ⊆ is the symbol for subsetting. Again, if you have not studied set theory you may safely ignore the table. The code below illustrates the behaviour of the above methods.

```
TreeSet<String> s = new TreeSet<String>();          // s is empty
s.add("dog"); s.add("cat"); s.add("rat");           // s: dog, cat, rat
HashSet<String> h = new HashSet<String>();          // h is empty
h.add("cow"); h.add("hen"); h.add("dog");           // h: cow, hen, dog
HashSet<String> w = new HashSet<String>(s);         // w: dog, cat, rat
System.out.print(s.isEmpty());                      // false appears
s.removeAll(h);                                     // s: cat, rat
h.addAll(w);                                        // h: cow, hen, dog, cat, rat
s.add("pig");                                       // s: cat, rat, pig
h.retainAll(s);                                     // h: cat, rat
System.out.print(s.containsAll(h));                 // true appears
System.out.print(s.containsAll(w));                 // false appears
```

Example 1: integers common to two sequences

The following program reads two (text) files of integers, and displays the integers that occur in both files. The numbers in each file need not be distinct, but multiple occurrences have no more significance than a single occurrence. For example, if the files contain, respectively,

```
11 10 23 13 45 23 34 67 56
```

and

```
34 88 23 51 67 76 13 10 80 29
```

then the program will output

```
[67, 34, 13, 23, 10]
```

(the order of elements here is not significant, and the square brackets are just a by-product of using Java's sets). Remember that integers can only be stored in sets of type `HashSet` or `TreeSet` if they are "wrapped" using class `Integer`; we will rely on autoboxing for this. The program is

```
import java.util.*;
import java.io.*;
class Commons {
    public static void main(String[] args) {
        // Read file 1 into set s1
        Scanner f1 = null;
        try {
            f1 = new Scanner(new File("numbers1.txt"));
        }
        catch(FileNotFoundException e) {
            System.out.println("File not found");
        }
        HashSet<Integer> s1 = new HashSet<Integer>();
        while (f1.hasNext()) {
            int k = f1.nextInt();
            s1.add(k); // using autoboxing
        }
        // Read file 2 into set s2
        Scanner f2 = null;
        try {
            f2 = new Scanner(new File("numbers2.txt"));
        }
        catch(FileNotFoundException e) {
            System.out.println("File not found");
        }
        HashSet<Integer> s2 = new HashSet<Integer>();
        while (f2.hasNext()) {
```

```
            int k = f2.nextInt();
            s2.add(k); // using autoboxing
        }
        // Compute & display intersection of s1 & s2
        s1.retainAll(s2);
        System.out.print(s1);
    }
}
```
■

We can use a for-each loop to access each element in a set, one after the other. For example, the following loop prints out all the elements of set `ws` (whose elements are strings):

```
for (String w: ws) {
    System.out.println(w);
}
```

If `ws` is of type `HashSet` then the order in which the elements appear is arbitrary. If `ws` is of type `TreeSet` they will appear in ascending order (again, this does not imply that the elements in the set have any inherent order – they don't).

3 Maps

A map is a collection of items where each item is composed of two parts, called a *key* and a *value*, respectively, such that all the keys are unique. For example, the following collection is a map, where each key (on the left) is the name of a person, and each value (on the right) is the person's age (the right-arrows are not part of Java, but just our way of visually connecting each key with its associated value):

Bill → 23
Jane → 25
Jack → 23
Ted → 29

Observe that the keys are unique, but that the associated values may contain duplicates. Keys are usually strings, and the associated values are frequently integers.

Java supplies two classes for maps: **`HashMap<T,U>`** and **`TreeMap<T,U>`**. They have similar interfaces, but different implementations. In typical applications `HashMap` is usually more efficient, unless processing the keys in sorted order is important. `T` stands for the type of keys, and `U` stands for the type of values. The map constructors are as follows:

HashMap()	HashMap(*Map*<*T,U*>)
TreeMap()	TreeMap(*Map*<*T,U*>)

We write `Map` above to indicate that the actual parameter may be an instance of `HashMap` or

`TreeMap`.

`new HashMap()` and **`new TreeMap()`** each creates an empty map. **`new HashMap(m)`** and **`new TreeMap(m)`** each creates a map with the same members as `m`. The methods in both classes are as follows:

```
void clear()
U put(T key, U value)
U get(Object key)
U remove(Object key)
boolean containsKey(Object key)
int size()
boolean isEmpty()
Set<T> keySet()
String toString()
```

`m.clear()` empties map `m`. **`m.put(key,val)`** inserts `key` → `val` in map `m`. If `m` previously contained `key`, its old associated value is replaced with `val`. The previous value associated with `key` is returned, or `null` if `key` was not present (the returned reference is typically ignored). **`m.get(key)`** returns the value associated with key `key` in map `m`, or `null` if `m` does not contain `key`. **`m.remove(key)`** removes from map `m` the key-value item whose key is `key`, if present. The previous value associated with `key` is returned, or `null` if `key` was not present. **`m.containsKey(key)`** returns true if map `m` contains key `key`. **`m.size()`** returns the number of key-value items in map `m`. **`m.isEmpty()`** returns `true` if map `m` is empty. **`m.keySet()`** returns a set of the keys in map `m`. We are not told whether the set returned is an instance of `HashSet` or `TreeSet`, but we can make good use of it by supplying it as the argument of a method or constructor that accepts any kind of set. For example, for `m` an instance of `HashMap<String,Integer>`, `new LinkedList<String>(m.keySet())` creates a list of type `LinkedList<String>` whose elements are precisely the keys in `m`. **`m.toString()`** returns a string representation of map `m`, consisting of a list of key-value pairs. The following illustrates some of the above methods:

```
HashMap<String,String> m =
            new HashMap<String,String> ();  // m is empty
m.put("Bill", "dog");                       // m: Bill→dog
m.put("Pam", "pony");                       // m: Bill→dog, Pam→pony
m.put("Joe", "cat");                        // m: Bill→dog, Pam→pony, Joe→cat
System.out.print(m.containsKey("Jim"));     // false appears
System.out.print(m.containsKey("dog"));     // false appears
System.out.print(m.get("Joe"));             // cat appears
System.out.print(m.get("Jim"));             // null appears
m.put("Pam", "dog");                        // m: Bill→dog, Pam→dog, Joe→cat
m.remove("Bill");                           // m: Pam→dog, Joe→cat
m.remove("Mary");                           // m: Pam→dog, Joe→cat
```

```
TreeSet<String> s =
            new TreeSet<String>(m.keySet());      // s: Pam, Joe
```

It is strongly recommended that you do not change the state of keys in a map; if you do, the effect on your program is unpredictable.

Example 1: a simple phone directory

We write a program which maintains a small telephone book. Each entry in the book consists of a person's name (a single word) and a number. A user at the keyboard can add new entries to the phone book, or look up a number. For example, the line

```
Billy 345656
```

causes the number `345656` to be associated with the name `Billy` in the phone book (if an entry already exists for `Billy`, it will be replaced). A command such as

```
Billy ?
```

will result in the program displaying the number associated with `Billy` in the phone book, or a message that there is no entry for `Billy` (we will allow any string in place of `?` provided it doesn't begin with a digit). Input of the end-of-file character terminates the program. The following is a typical dialogue at the keyboard:

```
Welcome to the phone book!
Billy 345656
Pat 76845
Billy ?
345656
Susan ?
Can't find Susan
Billy 654321
Billy ?
654321
```

In a professional program, the phone book would be saved when the program terminates but we won't worry about that here (if you care about that, see the chapter on serialization). The program is easy to write using a map to store the phone book:

```
import java.util.*;
class PhoneDirectory {
    public static void main(String args[]) {
        HashMap<String,String> directory = new HashMap<String,String> ();
                                                    // the phone book
        System.out.println("Welcome to the phone book!");
        while (!Console.endOfFile()) {
            String name = Console.readToken(); // read name
            String item = Console.readString(); // read number or "?"
```

```
            if (Character.isDigit(item.charAt(0)))  // add new number
                directory.put(name, item);
            else { // look up name
                String num = directory.get(name);
                if (num != null) System.out.println(num);
                else System.out.println("Can't find " + name);
            }
        }
    }
}
```

Example 2: word frequency count again

The following is an alternative solution to the problem of counting the frequency of words in a text file, as specified earlier. A typical invocation is

```
java FrequencyCount2 TheTempest.txt
```

after which the program prints the number of occurrences in text file `TheTempest.txt` of each word that a user enters at the keyboard. This time we use maps for a neater solution. As in the previous solution, the program consists of two phases. In the first phase, the program reads the file and builds a map whose keys are the words in the text file. Each key is associated with the number of times it occurs in the text. The second phase handles the user's queries by looking up in the map each word keyed in by the user.

```
import java.util.*;
import java.io.*;
class FrequencyCount2 {
    public static void main(String[] args) {
        // Phase 1: build a map of word-frequency items for each word in the file
        HashMap<String,Integer>  map = new HashMap<String,Integer> ();
                                                // for word-frequency items
        Scanner text = null;
        try {
            text = new Scanner(new File(args[0]));
        }
        catch(FileNotFoundException e) {
            System.out.println("File not found");
        }
        while (text.hasNext()) {
            String word = text.next();
            // make word lower case & remove any trailing punctuation mark
            word = word.toLowerCase();
            if (!Character.isLetterOrDigit(word.charAt(word.length()-1)))
                word = word.substring(0, word.length()-1);
```

```
            // update map with word
            if (!map.containsKey(word))  // new word: insert with frequency 1
                map.put(word, 1);  // using autoboxing
            else { // existing word: increment count
                int count =  map.get(word);  // old frequency
                map.put(word, count+1); // using autounboxing
            }
        }
        // Phase 2: process user's queries
        System.out.println("Enter words: ");
        while (!Console.endOfFile()) {
            String word = Console.readString();
            word = word.toLowerCase();
            int count; // for frequency count
            if (map.containsKey(word)) {
                count =  map.get(word); // using autounboxing
            }
            else count = 0;
            System.out.println("Frequency of " + word + ": " + count);
        }
    }
}
```

4 Equality and comparisons

Any object inserted into a collection must provide some or all of the methods `equals()`, `compareTo()`, and `hashCode()`, depending on the particular collection class. The table below gives a summary of what is sufficient.

	`equals()`	`compareTo()`	`hashCode()`
`TreeSet`, `TreeMap` (keys)		✓	
`HashSet`, `HashMap` (keys)	✓		✓
`ArrayList`, `LinkedList`	✓		

For example, if an object of class `CC`, say, is to be inserted into a collection of type `HashSet` then `CC` must include methods `equals()` and `hashCode()`, and if an object of class `CC` is to be a key in a map of type `TreeMap`, then `CC` must include method `compareTo()`. See the chapter on lists for how to provide `equals()` and `compareTo()`; we deal with `hashCode()` below.

`HashSet` treats objects of programmer-defined classes as distinct when they have distinct references, even if an `equals()` based on contents is defined. For example, execution of the code

```
HashSet<Person> set = new HashSet<Person>();
set.add(new Person("Bill", 23));
System.out.println(set.contains(new Person("Bill", 23)));
```

will cause `false` to be output, even if `Person` includes an `equals()` based on contents rather than references. The implementation of `HashSet` compares objects using a method called `hashCode()` which is supplied by default with every object. An invocation of `p.hashCode()` returns an integer representing object `p`. If for objects `p` and `q`, `p.hashCode()` and `q.hashCode()` return different integers, then objects `p` and `q` are taken to be distinct. Only when the values returned are equal is a real test for equality carried out. The default version of `hashCode()` behaves poorly because the integer returned depends on the object's reference rather than its contents. Hence objects with different references are treated as distinct, as in the example above. For any class whose instances may be stored in a set of type `HashSet`, the class must redefine `hashCode()` so that it returns an integer based solely on the values of its instance variables. Classes supplied by Java such as `Integer` and `String` redefine `hashCode()` appropriately. The following is an example of a suitable `hashCode()` to be added to a class `Person` with instance variables representing a person's name (`name`) and age (`age`):

```
public int hashCode() {
   return name.hashCode() + 3*age;
}
```

The header must always be `public int hashCode()`. The integer returned should be derived from the values of the instance variables, computed in any way you chose. In this case, we use the `hashCode()` method in the `String` class to get an integer from `name`, to which we add three times the value of `age`. We might just as well have written, say:

```
public int hashCode() {
   return 11*name.hashCode() + 7*age;
}
```

In the case of instance variables that are not integers, it saves work to make use of their `hashCode()` function, as in `name.hashCode()` above. All the primitive wrapper classes redefine `hashCode()` appropriately. For example, to derive an integer from a double `x` you can use `(new Double(x).hashCode())`.

As far as correct behaviour is concerned, it doesn't matter what integer is returned by `hashCode()` as long as it is compatible with `equals()`. By "compatible" we mean that whenever `p.equals(q)` yields `true`, `p.hashCode()` and `q.hashCode()` yield the same value. You must also include a definition of `equals()` based on the contents of the object, as we have done for class `Person`. For the best performance, you should try to design `hashCode()` so that a change in the value of an instance variable is likely to change the value returned by `hashCode()`. For example, the following is a poor design of `hashCode()` for

class `Person` above:

```
public int hashCode() {
   return age;
}
```

It is poor because a change in `name` has no effect on the value returned by `hashCode()`.Your program will still work, but not efficiently. Your program will even work if `hashCode()` returns a constant:

```
public int hashCode() {
   return 1;
}
```

– but it certainly won't execute quickly.

The class which provides the keys in `HashMap` must define method `hashCode()`.

5 Large example: constructing a line index OPTIONAL

To illustrate the use of the collection classes in an object-oriented design, we write a program which constructs an index for a text. A typical invocation is

```
java MakeIndex source.txt indexwords.txt
```

The arguments are the names of text files; the first one is called the *source* file and the second one is called the *dictionary* file. The program produces an alphabetical list of the words in the source file, each word accompanied by the line numbers in which it occurred, but listing only those words that occur in the dictionary file. For example, suppose the source and dictionary files are as follows:

```
I do not like thee, Doctor Fell,
The reason why, I cannot tell;
But this I know, and know full well,
I do not like thee, Doctor Fell.
```

```
sausages
fell
know
scissors
excel
doctor
reason
```

Then the output is:

```
doctor              1 4
fell                1 4
know                3
reason              2
```

For example, there is an entry for `fell` in the output because it occurs in the source file (on lines 1 and 4) and is included in the dictionary file. The output is aligned in two columns, one for words and one for line numbers. The line numbers associated with each word are in ascending order with no duplicates. The words in the dictionary are in lower case, one word per line (without any blanks on the line). Case is not treated as significant in the source text.

This is a challenging exercise not just in the use of the collection classes, but in object-oriented design. You shouldn't expect to take it all in in one reading. Make a light pass over it first to understand the general structure, and then revisit it once or twice to take in more detail. If you have come to the collection classes early in your study of Java, you may prefer to omit the example until you have gained more experience of object-oriented programming.

We will have to look up the dictionary for every word in the source file, and so it is advantageous to construct a memory-resident version as a set. `HashSet` is tailor-made for dictionaries:

```
import java.util.*;
import java.io.*;
class Dictionary {
    private HashSet<String> dictionary; // the dictionary as a set of words

    Dictionary (String fileName) { //initialise dictionary with words in file fileName
        try {
            dictionary = new HashSet<String>();
            Scanner in = new Scanner(new File (fileName));
            while (in.hasNextLine()) {
                String word = in.nextLine();
                dictionary.add(word);
            }
            in.close();
        }
        catch(FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    boolean contains(String w) { // Is w in dictionary, ignoring case?
        return dictionary.contains(w.toLowerCase());
    }
```

```
}
```

A `Dictionary` object is constructed by supplying a file whose words will be inserted in the dictionary. It sole method `contains()` determines whether a word occurs (in lower case form) in the dictionary.

As regards the source file, it will surely be convenient if we can access it using a method which delivers both the next word in the file and the line number in which it occurs. For that we will need a trivial class `WordItem` which encapsulates a word-number pair.

```
class WordItem {

    private String word; // a word
    private int lineNum; // a line number

    WordItem(String w, int n) { // word w, line number n
        word = w; lineNum = n;
    }

    String theWord() {
        return word;
    }

    int theLineNum() {
        return lineNum;
    }
}
```

Our intention now is to organise the input so that it delivers a stream of word-number pairs. The number attached to each word is the number of the line in which the word occurs in the source file. Let us call a sequence of word-number pairs a *word-number stream*, and encapsulate it in class `WordNumberStream` explained below.

```
class WordNumberStream {

    private Scanner in; // input text
    private Scanner thisLine = new Scanner("");  // the current line
    private int thisLineNum = 0; // current line number

    WordNumberStream(String fileName)  {
    // Associate text file fileName with this word-number stream
        try {
            in = new Scanner(new File(fileName));
        }
        catch(FileNotFoundException e) {
```

```
            in = null; // no point, but compiler insists
            e.printStackTrace();
        }
    }

    WordItem getItem() {
    // Return next word and its line number, or null if no more words
        while (!thisLine.hasNext()) {  // current line exhausted
            if (!in. hasNextLine())
                return null; // no more input
            String buffer = in.nextLine();
            thisLineNum++;
            thisLine = new Scanner(buffer);
        }
        String word = thisLine.next(); // next word in input
        // make word lower case & remove any trailing punctuation mark
        word = word.toLowerCase();
        if (!Character.isLetterOrDigit(word.charAt(word.length()-1)))
            word = word.substring(0, word.length()-1);
        return new WordItem(word, thisLineNum);
    }
}
```

The constructor does no more than open the source file. Method `getItem()` returns the next word in the input accompanied by its line number. It needs to access the input both line-by-line (to keep track of line numbers) and word-by-word to retrieve the individual words. Hence we employ two `Scanner` objects: `in` handles line-by-line input from the source file and `thisLine` handles the words on the current line. In general, the words yet to be indexed are those remaining in `thisLine` followed by those remaining in the source file. The words in `thisLine` are taken from line number `thisLineNum` in the source file. When `thisLine` is exhausted, we refill it with the next line from the input (and increment `thisLineNum`). The while-statement at the start of `getItem()` is needed to accommodate the possibility that a run of lines may have no tokens (a line might well contain nothing but spaces).

That takes care of the input side. On the output side, we have to generate for each word a sequence of line numbers in ascending order. Let us call such a sequence an *up-sequence* and encapsulate it in class `UpSequence`:

```
class UpSequence {

    private LinkedList<Integer> seq; // a non-empty up-sequence

    UpSequence(int n) { // Create an up-sequence whose only element is n
        seq = new LinkedList<Integer>();
        seq.add(n);
```

```
    }

    void append(int n) {
    // append n if not present, given n is no smaller than current elements
        int lastInt = seq.getLast(); // last integer in seq
        if (n!=lastInt)  // n > lastInt & hence is new
            seq.add(n);
    }

    void put() { // display up-sequence on a line, items separated by a blank
        for (Integer k: seq) {
            System.out.print(k + " ");
        }
        System.out.println();
    }
}
```

In `UpSequence`, the line numbers are stored in a list of type `LinkedList`. We have chosen this in preference to `ArrayList` because it provides the convenience of method `getLast()`. There is no difficulty in keeping the sequence in ascending order, because that is the way the line numbers naturally arrive. It also makes it easy for `append()` to avoid entering duplicates: if a newly-arriving line number already occurs in the sequence it can only be in the final position, and that is easy to inspect.

Each item in the final output consists of a word accompanied by an up-sequence. Let us call a collection each of whose elements consist of a word and an up-sequence a *word index*, and encapsulate it in class `WordIndex` given below. `WordIndex` has two methods: one to append a new line number to the up-sequence associated with a given word, and one to print the word index in its entirety.

```
class WordIndex {

    private HashMap<String,UpSequence> index =
                                        new HashMap<String,UpSequence>();

    void addEntry(WordItem wi) {
    // add wi.theLineNum() to the up-sequence associated with wi.theWord()
    // (wi.theLineNum() is  no smaller than current entries in the up-sequence)
        UpSequence lineNums = index.get(wi.theWord());
        if (lineNums == null)  // first occurrence of wi.theWord()
            index.put(wi.theWord(), new UpSequence(wi.theLineNum()));
        else // wi.theWord() is present and its associated UpSequence is lineNums
            lineNums.append(wi.theLineNum());
    }
```

```
    void put() {
    // display index in asscending order by word, formatted as already described
        ArrayList<String> words = new ArrayList<String>(index.keySet());
                                                                    // the words
        Collections.sort(words);
        for (String w: words) {
            // print index entry for a single word
            UpSequence lineNumbers = index.get(w); // line numbers
            System.out.print(w + "                    ".substring(w.length()));
                                          // print word padded to length 20
            lineNumbers.put(); // print line numbers
        }
    }
}
```

List `words` in `put()` has been chosen to be of type `ArrayList`, but `LinkedList` would have been equally good.

Finally, we write class `MakeIndex`. This contains just method `main()` which constructs and displays the index for given source and dictionary files:

```
class MakeIndex {
    public static void main(String[] args) {
        WordNumberStream stream = new WordNumberStream(args[0]);
        Dictionary dictionary = new Dictionary(args[1]);
        WordIndex wordIndex = new WordIndex(); // word index (initially empty)
        WordItem wordItem = stream.getItem();  // first word
        while (wordItem !=null) {
            if (dictionary.contains(wordItem.theWord()))
                wordIndex.addEntry(wordItem);
            wordItem  = stream.getItem();
        }
        wordIndex.put();
    }
}
```