HashMap Example

Week 4 Presentation 4

Maps

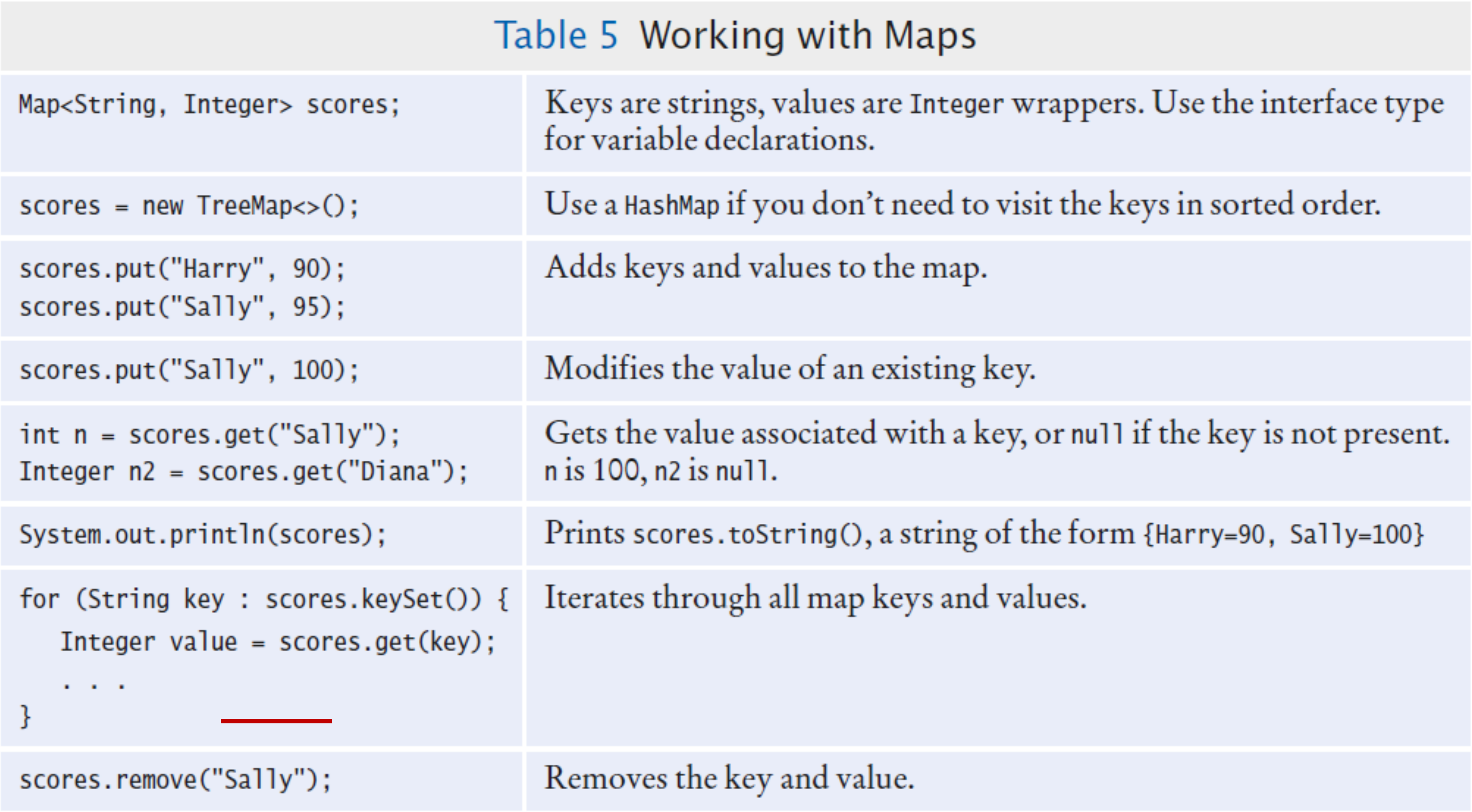


Table 5 Working with Maps

`Map<String, Integer> scores;`	Keys are strings, values are `Integer` wrappers. Use the interface type for variable declarations.
`scores = new TreeMap<>();`	Use a `HashMap` if you don't need to visit the keys in sorted order.

`scores.put("Harry", 90);`
`scores.put("Sally", 95);`
	Adds keys and values to the map.

`scores.put("Sally", 100);`	Modifies the value of an existing key.

`int n = scores.get("Sally");`
`Integer n2 = scores.get("Diana");`
	Gets the value associated with a key, or `null` if the key is not present. `n` is 100, `n2` is `null`.

`System.out.println(scores);`	Prints `scores.toString()`, a string of the form `{Harry=90, Sally=100}`

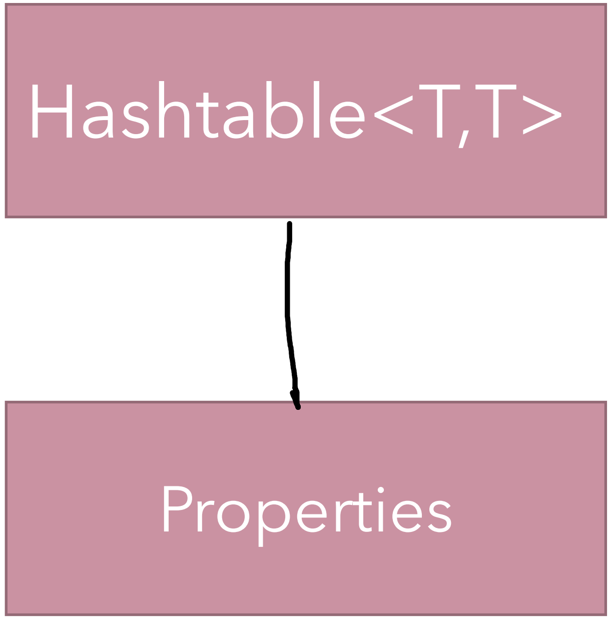
`for (String key : scores.keySet()) {`
`   Integer value = scores.get(key);`
`   . . .`
`}`
	Iterates through all map keys and values.

`scores.remove("Sally");`	Removes the key and value.

Properties

- One interesting (and very useful) example of hash tables is the Properties class, which associates two strings.
- java.util.Properties

- java.lang.Object
  - java.util.Dictionary<K,V>
    - java.util.Hashtable<Object,Object>
      - java.util.Properties

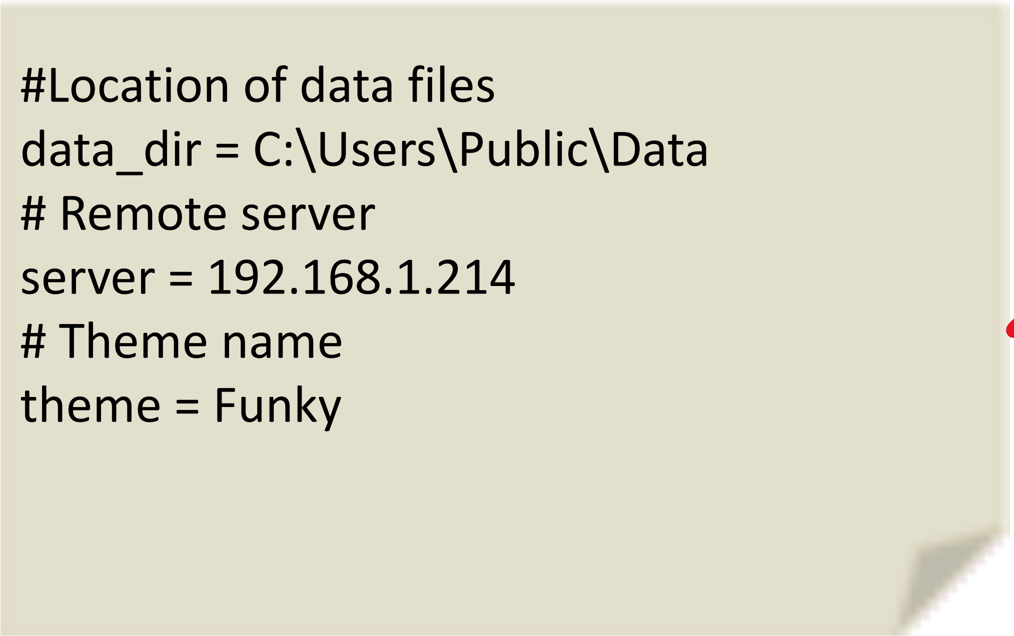


Key: String

Value: String

Example 1

Properties

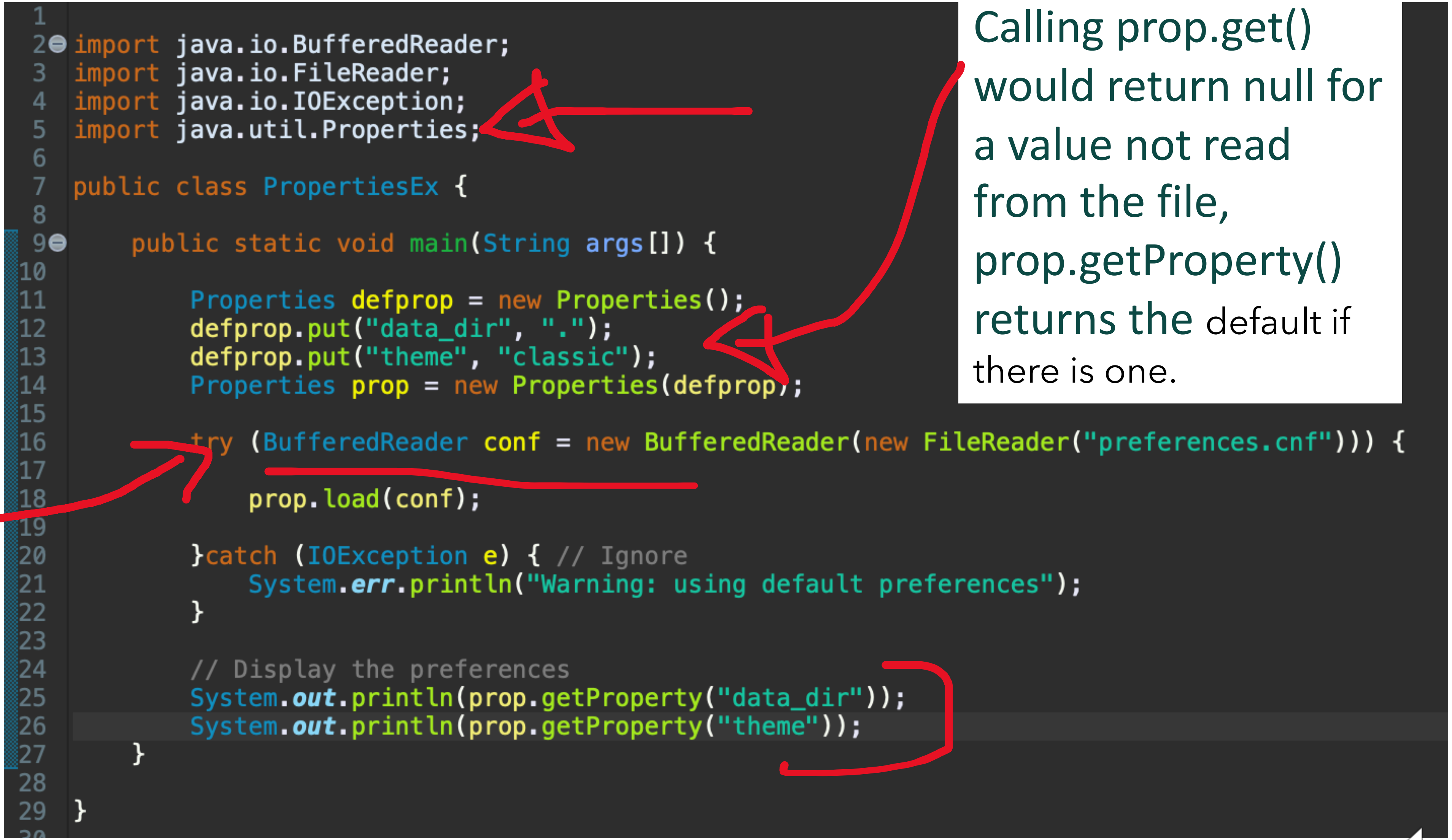


```
#Location of data files
data_dir = C:\Users\Public\Data
# Remote server
server = 192.168.1.214
# Theme name
theme = Funky
```

preferences.cnf

When you install a piece of software on your computer, you are usually asked a lot of questions, such as where you want to install the program, the location of other resources, possibly a theme. All this information is stored in a .ini, .conf or whatever file. Each parameter is a name associated with a value. Properties objects deal with these files, can read them, write them, and ignore for instance lines starting with #.

You can define default values. Calling prop.get() would return null for a value not read from the file, prop.getProperty() returns the default if there is one.



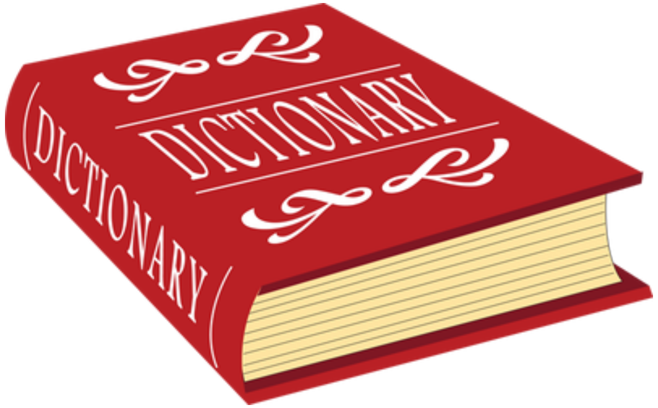
```java

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Properties;

public class PropertiesEx {

    public static void main(String args[]) {

        Properties defprop = new Properties();
        defprop.put("data_dir", ".");
        defprop.put("theme", "classic");
        Properties prop = new Properties(defprop);

        try (BufferedReader conf = new BufferedReader(new FileReader("preferences.cnf"))) {

            prop.load(conf);

        }catch (IOException e) { // Ignore
            System.err.println("Warning: using default preferences");
        }

        // Display the preferences
        System.out.println(prop.getProperty("data_dir"));
        System.out.println(prop.getProperty("theme"));
    }

}
```

Try with resources. Remember?

Example 2

A *multimap* is like a Map but it can map each key to multiple values.

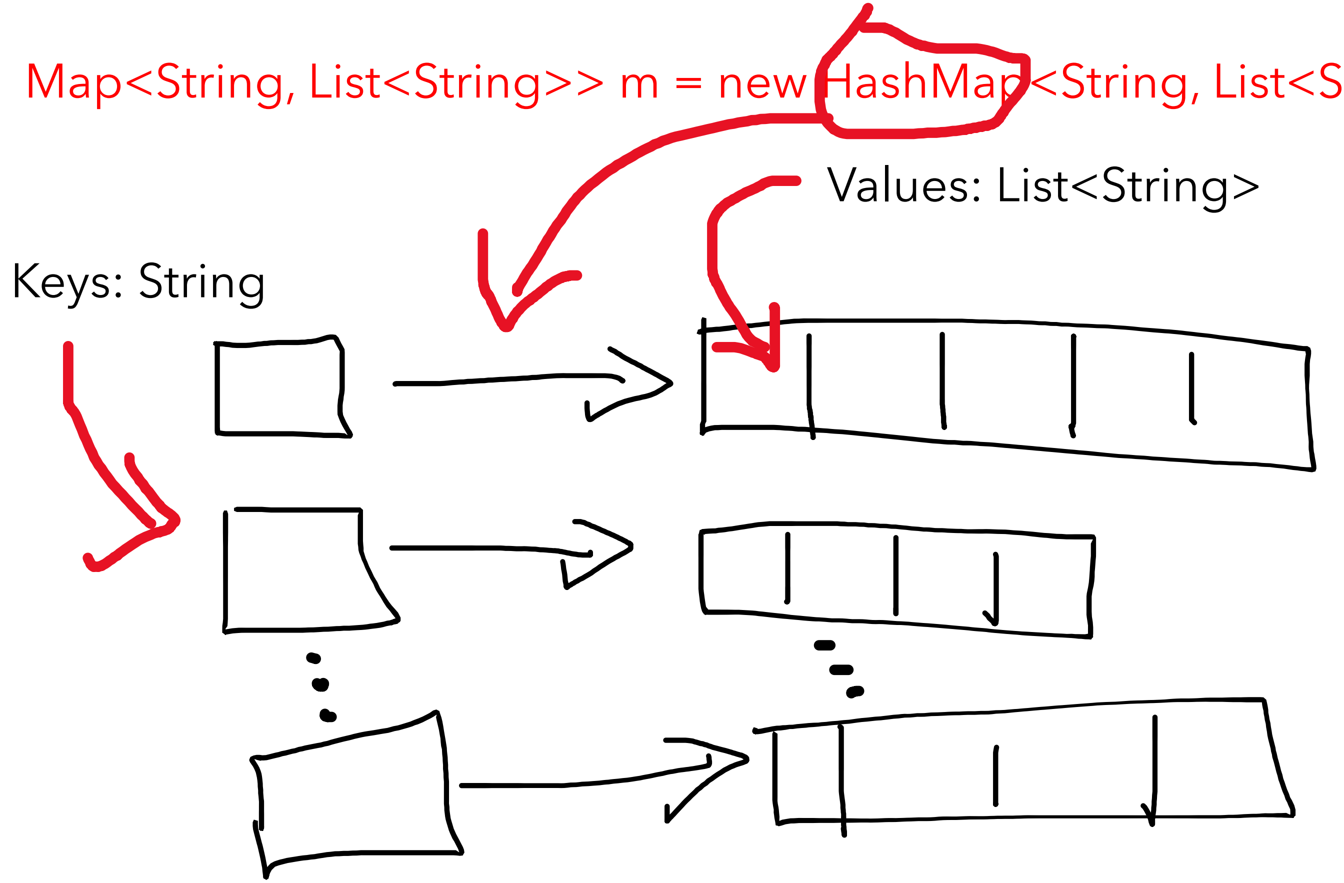
Problem: how can we find all the anagrams of every word in a dictionary?



Anagrams are words that are created by rearranging the letters in a word – for example “late” is an anagram of “tale”.

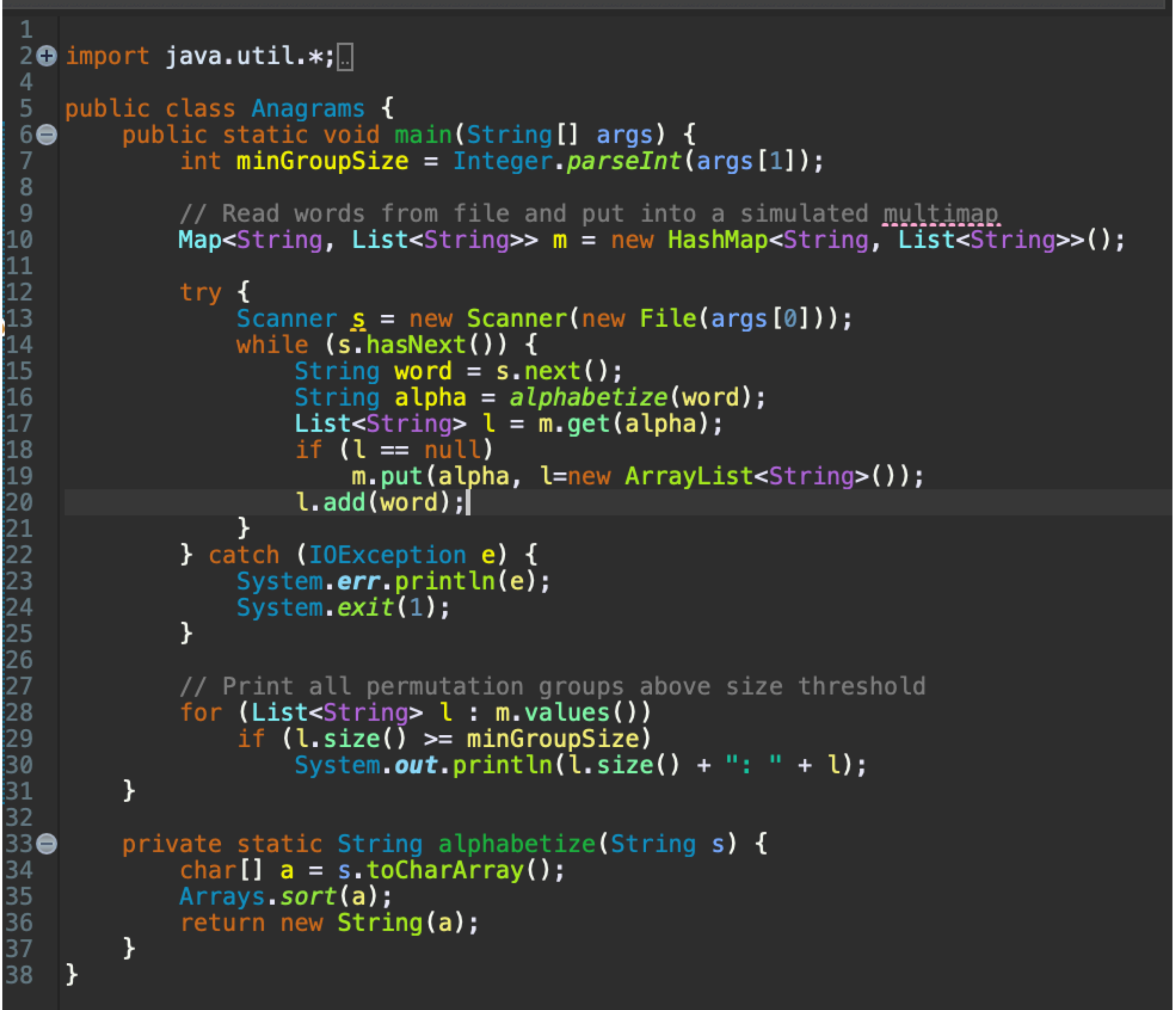
A *multimap* is like a Map but it can map each key to multiple values.

Map<String, List<String>> m = new HashMap<String, List<String>>();



Values: List<String>

Keys: String



```java
import java.util.*;

public class Anagrams {
    public static void main(String[] args) {
        int minGroupSize = Integer.parseInt(args[1]);

        // Read words from file and put into a simulated multimap
        Map<String, List<String>> m = new HashMap<String, List<String>>();

        try {
            Scanner s = new Scanner(new File(args[0]));
            while (s.hasNext()) {
                String word = s.next();
                String alpha = alphabetize(word);
                List<String> l = m.get(alpha);
                if (l == null)
                    m.put(alpha, l=new ArrayList<String>());
                l.add(word);
            }
        } catch (IOException e) {
            System.err.println(e);
            System.exit(1);
        }

        // Print all permutation groups above size threshold
        for (List<String> l : m.values())
            if (l.size() >= minGroupSize)
                System.out.println(l.size() + ": " + l);
    }

    private static String alphabetize(String s) {
        char[] a = s.toCharArray();
        Arrays.sort(a);
        return new String(a);
    }
}
```

There is a standard trick for finding anagram groups: For each word in the dictionary, alphabetize the letters in the word (that is, reorder the word's letters into alphabetical order) and put an entry into a multimap, mapping the alphabetized word to the original word.

For example, the word *bad* causes an entry mapping *abd* into *bad* to be put into the multimap.

A moment's reflection will show that all the words to which any given key maps form an anagram group. It's a simple matter to iterate over the keys in the multimap, printing out each anagram group that meets the size constraint.

```
> java Anagrams "../dictionary.txt" 5
```

```
5: [dates, sated, stade, stead, tsade]
6: [dearths, hardest, hardset, hatreds, threads, trashed]
6: [dater, derat, rated, tared, trade, tread]
5: [entasis, nasties, sestina, tansies, tisanes]
5: [actors, castor, costar, scrota, tarocs]
5: [delist, idlest, listed, silted, tildes]
5: [intreat, iterant, nattier, nitrate, tertian]
9: [anestri, antsier, nastier, ratines, retains, retinas, retsina,
stainer, stearin]
```

See: https://docs.oracle.com/javase/tutorial/collections/interfaces/map.html

Example 3

Another example uses a Tokenizer class that counts words from a text file one by one, ignoring space and punctuation.

Goal: finding the 10 most used words in a speech.

We need to associate with each word a counter (so, we need a Map<String,Integer>). If we don't know the word, we store it with "1". Otherwise, we retrieve the counter, increase its value by one, and store it back.

When we have counted words, we need to find the 10 most used − we can use a map (associating a number of occurrences to a list of words, as there may be ties) but we also need some ordering. We need to go through our hash map and store its objects in, for instance, a TreeMap<Integer,TreeSet<String>>. Then we can iterate on the tree map and retrieve the most common words.

The result is usually very disappointing, because in an English speech the most common words are likely to be "the", "is", "a", and so forth. Those words are not very significant words and are usually called "stop words" (search engines on the Internet ignore them).
What we need to do is have a list of stop words, read it into an easily searchable structure such as a tree, and start counting words only when we cannot find them in this list of not important words. It gives a completely different vision of a speech.

Exercise: implement this algorithm.