Exercise

• Write a program to count the occurrences of each word in a text file (e.g. Moby Dick or Green Eggs and Ham).
  – Allow the user to type a word and report how many times that word appeared in the book.
  – Report all words that appeared in the book at least 500 times, in alphabetical order.

• How will we store the data to solve this problem?

The Map ADT

• **map**: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.

• basic map operations:
  – **put**(key, value): Adds a mapping from a key to a value.
  – **get**(key): Retrieves the value mapped to the key.
  – **remove**(key): Removes the given key and its reference to the mapped value.

```
myMap.get("Juliet") returns "Capulet"
```

Map implementation

• in Java, maps are represented by **Map interface** in java.util

• Map is implemented by the **HashMap** and **TreeMap** classes
  – **HashMap**: implemented using an array called a "hash table"; extremely fast: **O(1)**; keys are stored in unpredictable order
  – **TreeMap**: implemented as a linked "binary tree" structure; very fast: **O(log N)**; keys are stored in sorted order
  – A map requires 2 type parameters: one for keys, one for values.

```
// maps from String keys to Integer values
Map<String, Integer> votes = new HashMap<String, Integer>();
```
Map methods

- `put(key, value)` adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one.
- `get(key)` returns the value mapped to the given key (null if not found).
- `containsKey(key)` returns true if the map contains a mapping for the given key.
- `remove(key)` removes any existing mapping for the given key.
- `clear()` removes all key/value pairs from the map.
- `size()` returns the number of key/value pairs in the map.
- `isEmpty()` returns true if the map's size is 0.
- `toString()` returns a string such as `{a=90, d=60, c=70}`.
- `keySet()` returns a set of all keys in the map.
- `values()` returns a collection of all values in the map.
- `putAll(map)` adds all key/value pairs from the given map to this map.
- `equals(map)` returns true if given map has the same mappings as this one.

Using maps

- A map allows you to get from one half of a pair to the other.
  - Remembers one piece of information about every index (key).
    
    ```java
    // key value
    put("Daniel", "541-346-4154")
    ```

    - Later, we can supply only the key and get back the related value: Allows us to ask: What is Daniel’s phone number?
      
      ```java
      Map.get("Daniel")
      ```
      

Exercise solution

```java
// read file into a map of [word --> number of occurrences]
Map<String, Integer> wordCount = new HashMap<String, Integer>();
Scanner input = new Scanner(new File("hamlet.txt"));
input.useDelimiter("[^a-zA-Z\']+");
while (input.hasNext()) {
    String word = input.next().toLowerCase();
    if (wordCount.containsKey(word)) {  // seen this word before; increase count by 1
        int count = wordCount.get(word);
        wordCount.put(word, count + 1);
    } else {  // never seen this word before
        wordCount.put(word, 1);
    }
}
Scanner console = new Scanner(System.in);
System.out.print("Word to search for? ");
String word = console.next().toLowerCase();
System.out.println("appears "+ wordCount.get(word) + " times.");
```

keySet and values

- `keySet()` method returns a set of all keys in the map
  - can loop over the keys in a foreach loop
  - can get each key’s associated value by calling `get` on the map

```java
Map<String, Integer> ages = new HashMap<String, Integer>();
ages.put("Marty", 19);
ages.put("Geneva", 2);
ages.put("Vicki", 57);
for (String name : ages.keySet()) {  // Geneva -> 2
    int age = ages.get(name);
    System.out.println(name + " -> " + age);
    // Marty -> 19
    // Vicki -> 57
}
```

- `values()` method returns a collection of all values in the map
  - can loop over the values in a foreach loop
  - there is no easy way to get from a value to its associated key(s)
Maps/Dictionaries are a built-in feature of many scripting languages!

**Perl**:
```perl
my %wordcount = ();
...
if (defined($wordCount{$word})) {
    $wordCount{$word}++;
} else {
    $wordCount{$word} = 1;
}
```

**Python**:
```python
wordCount = {}
...
if wordCount.has_key(word):
    wordCount[word] += 1
else:
    wordCount[word] = 1
```

**Languages and Grammars**

- **(formal) language**: A set of words or symbols.
- **grammar**: A description of a language that describes which sequences of symbols are allowed in that language.
  - describes language *syntax* (rules) but not *semantics* (meaning)
  - can be used to generate strings from a language, or to determine whether a given string belongs to a given language

**Backus-Naur (BNF)**

- **Backus-Naur Form (BNF)**: A syntax for describing language grammars in terms of transformation *rules*, of the form:

  `<symbol> ::= <expression> | <expression> ... | <expression>`

  - **terminal**: A fundamental symbol of the language.
  - **non-terminal**: A high-level symbol describing language syntax, which can be transformed into other non-terminal or terminal symbol(s) based on the rules of the grammar.

  - Java grammar
An example BNF grammar

<s>::=<n> <v>
<n>::=Marty | Stuart | Victoria | Watson
<v>::=cried | slept | won Jeopardy

- Some sentences that could be generated from this grammar:
  Marty won Jeopardy
  Stuart cried
  Watson slept

BNF grammar version 2

<s>::=<np> <v>
<np>::=<pn> | <dp> <n>
<pn>::=Marty | Stuart | Victoria | Watson
<dp>::=a | the
<n>::=ball | hamster | carrot | computer
<v>::=cried | slept | won Jeopardy

- Some sentences that could be generated from this grammar:
  the carrot cried
  Watson won Jeopardy
  a computer slept

BNF grammar version 3

<s>::=<np> <v>
<np>::=<pn> | <dp> <adj> <n>
<pn>::=Marty | Stuart | Victoria | Watson
<dp>::=a | the
<adj>::=silly | invisible | loud | romantic
<n>::=ball | hamster | carrot | computer
<v>::=cried | slept | won Jeopardy

- Some sentences that could be generated from this grammar:
  the invisible carrot cried
  Watson won Jeopardy
  a computer slept
  a romantic ball won Jeopardy

Grammars and recursion

<s>::=<np> <v>
<np>::=<pn> | <dp> <adjp> <n>
<pn>::=Marty | Stuart | Victoria | Watson
<dp>::=a | the
<adjp>::=<adj> <adjp> | <adj>
<adj>::=silly | invisible | loud | romantic
<n>::=ball | hamster | carrot | computer
<v>::=cried | slept | won Jeopardy

- Grammar rules can be defined recursively, so that the expansion of a symbol can contain that same symbol.
  - There must also be expressions that expand the symbol into something non-recursive, so that the recursion eventually ends.
Grammar, final version

<s>::=<np> <vp>

<np>::=<dp> <adjp> <n>|<pn>
<dp>::=the|a
<adjp>::=<adj>|<adjp> <adjp>
<adj>::=big|fat|green|wonderful|faulty|subliminal
<n>::=dog|cat|man|university|father|mother|child
<pn>::=<John|Jane|Sally|Spot|Fred|Elmo
<vp>::=<tv> <np>|<iv>
<tv>::=hit|honored|kissed|helped
<iv>::=died|collapsed|laughed|wept

• Could this grammar generate the following sentences?
  Fred honored the green wonderful child
  big Jane wept the fat man fat
• Generate a random sentence using this grammar.