fish.txt

```
One fish
two fish
red fish
blue fish.

Black fish
blue fish
old fish
new fish.

This one has
a little star.

This one has a little car.
Say! What a lot
of fish there are.
```

fishIndex.txt

```
A 12, 14, 15
ARE 16
BLACK 6
BLUE 4, 7
CAR 14
FISH 1, 2, 3, 4, 6, 7, 8, 9, 16
HAS 11, 14
LITTLE 12, 14
LOT 15
NEW 9
OF 16
OLD 8
ONE 1, 11, 14
RED 3
SAY 15
STAR 12
THERE 16
THIS 11, 14
TWO 2
WHAT 15
```

Figure 1 A sample text file and its index

The *Index Maker* program consists of three classes (Figure 2). It also uses `ArrayList` in two ways: `IndexEntry` has an `ArrayList<Integer>` field that holds the line numbers, and `DocumentIndex` extends `ArrayList<IndexEntry>`.

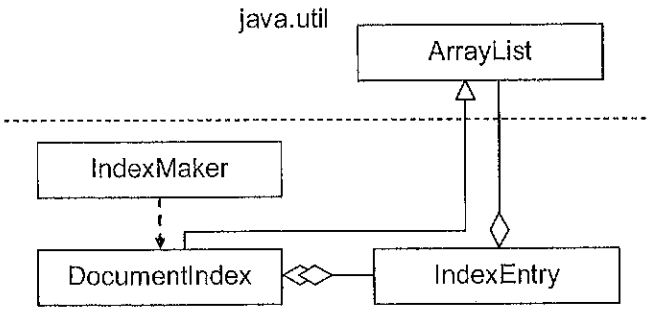

Figure 2 *Index Maker* classes

The `IndexMaker` class is the main class. We have provided this class for you on Haiku. Its `main` method prompts the user for the names of the input and output files (or obtains them from command-line arguments, if supplied), opens the input file, creates an output file, reads and processes all the lines from the input file, then saves the resulting document index in the output file.

Writing the `DocumentIndex` and `IndexEntry` classes is left to you (possibly in a team with another programmer). You don't have to deal with reading or writing files in this lab.

The `IndexEntry` class

An `IndexEntry` object represents one index entry. It has two fields:

```
private String word;
private ArrayList<Integer> numsList;
```

The numbers in `numsList` represent the line numbers where `word` occurs in the input file. (Note that the `IndexEntry` class is quite general and reusable: the numbers can represent line numbers, page numbers, etc., depending on the application.)

Provide a constructor for this class that takes a given word (a `String`), converts it into the upper case (by calling `toUpperCase`), and saves it in `word`. The constructor should also initialize `numsList` to an empty `ArrayList<Integer>`.

This class should have the following three methods:

1. `void add(int num)` — appends `num` to `numsList`, but only if it is not already in that list. (The `ArrayList`'s `contains` method expects an object as a parameter, and the `add` method expects an `Integer`, but `num` can be converted into an `Integer` automatically, due to autoboxing.)
2. `String getWord()` — this is an accessor method; it returns `word`.
3. `String toString()` — returns a string representation of this `IndexEntry` in the format used in each line of the output file (Figure 1).

The `DocumentIndex` class

A `DocumentIndex` object represents the entire index for a document: the list of all its index entries. The index entries should always be arranged in alphabetical order, as shown in Figure 1.

Make the `DocumentIndex` class extend `ArrayList<IndexEntry>`. Provide two constructors: one that creates a list with the default capacity, the other that creates a list with a given capacity. (These constructors simply call the respective constructors of the superclass, `ArrayList`.)

`DocumentIndex` should have the following two public methods:

1. `void addWord(String word, int num)` — adds `num` to the `IndexEntry` for `word` by calling that `IndexEntry`'s `add(num)` method. If `word` is not yet in this `DocumentIndex`, the method first creates a new `IndexEntry` for `word` and inserts it into this list in alphabetical order (ignoring the upper and lower case).
2. `void addAllWords(String str, int num)` — extracts all the words from `str` (skipping punctuation and whitespace) and for each word calls `addWord(word, num)`.

   You could code the word extractor yourself, of course, but it is much better to use the `String` class's `split` method. Look it up in the Java API. Use the one that takes one parameter, `regex`, that is, a *regular expression*[regex]. Regular expressions are not specific to Java: they are used in many languages and text parsers. `regex` describes the match pattern for all possible word separators. Use `"\\W+"` here. `\W` (with an uppercase 'W') stands for any "non-word" character, that is, any character that is not a digit or a letter. + means "occurs at least once." (Regular expressions use backslash as the escape character; hence the double backslash in the literal string.)

   `split` returns an array of `String`s. Use a "for each" loop to call `addWord` for each word in that array. Note, however, that `split` may put an empty string into the resulting array — when `str` starts with a separator or when `str` is empty. This is an unfortunate decision (or a bug). Make sure you skip empty strings and do not call `addWord` for them.

We recommend that you also define a private helper method

```
private int foundOrInserted(String word)
```

and call it from `addWord`. This method should traverse this `DocumentIndex` and compare `word` (case-blind) to the words in the `IndexEntry` objects in this list, looking for the position where `word` fits in alphabetically. If an `IndexEntry` with `word` is not already in that position, the method creates and inserts a new `IndexEntry` for `word` at that position. The method returns the position (we'd like to say "the index" but we have too many indices going already!) of the either found or inserted `IndexEntry`.

Test your program thoroughly on different text data files, including an empty file, a file with blank lines, a file with lines that have leading spaces or punctuation, a file with multiple occurrences of a word on the same line, and a file with the same word on different lines.