Compuer Science 62
Assignment 2

Due 11:59pm on Tuesday, February 9, 2009

A common task for many text processing applications is to collect a frequency count of all of the words in a text document, i.e. many times each word occurs in the document. For our purposes, a *word* is a sequence of two or more letters (i.e. letters of the alphabet) separated by non-letters. We will not distinguish between upper- and lower-case letters, so *Mary* and *mary* will be treated as the same word.

Once we've counted how many time each word occurs, we'll then print it out using our new found formatting prowess. Words will be printed (and stored) in alphabetical order. For example, given the text *"I Am what I am and that's all that I am"*, we would get the following output:

```
all                     1
am                      3
and                     1
that                    2
what                    1
```

Notice that all the words are lowercase, that *am* and *Am* have been combined and that *that's* was split into two parts and *s* only contained on letter, so it did not count as a *word*.

1 The work

Getting Started: To get you started, I've provided you with two classes, `Pair` similar to the one we designed in class and `WordScanner` that will help us with the text processing (see the description at the end of this document) as well as a skeleton class you'll need to fill in `SortedPairs`. As before, create a new project and call it something like "Assignment2". Copy the `Pair`,

`WordScanner` and `SortedPairs` classes into your source directory from `/common/cs/cs062/assignments/assignment2/` and select "File/Refresh".

1.1 BetterInteger

Your first task is to construct a `BetterInteger` class. To create a new class, select "File/New/Class" and fill in the appropriate information. The problem with the built-in `Integer` class is that once you construct the object, you cannot modify it. Your `BetterInteger` class should support the following functionality:

- constuct a new BetterInteger given an integer value
- getValue and setValue methods for getting and setting the integer value
- an increment method to increase the value of the stored integer by 1
- an addValue method that increased the value of the stored intger by the value passed in

1.2 SortedPairs

Now, you need to fill in the details for the `SortedPairs` class. While for our current assignment we'll only need to store `Pair`s containing `String`s and `BetterInteger`s, instead of only being able to use `String`s, we're going to build it so we can store (and therfore count) any object type that implements the `Comparable` interface (i.e. has `compareTo` and `equals` implemented appropriately, of which `String` is one). To accomplish this, the method header for the `SortedPairs` class uses generics:

```
public class SortedPairs <E extends Comparable<E>>
```

This states that we have a generic type called `E` that `extends` some class that implements the `Comparable` interface for type `E`, i.e. the method `compareTo(E obj)`. In our case you would say something like:

```
SortedPairs<String> s = new SortedPairs<String>();
```

You will need to add the following methods to the `SortedPairs` class (use exactly these method names):

- add: which takes a single parameter and adds that item to our data structure. If we already have it, we should increment the count. Items should be added and stored *in alphabetical order*.

- print: which takes a single parameter which is a `PrintStream` and prints out the stored data to the stream using the `format` method. The data should be printed one word per line in alphabetical order. The word should be left aligned and use 20 characters and the frequency of the word should be right aligned and use 5 characters (see the printed example above).

- buildAndPrint: which takes a `File` and `PrintStream` as parameters. This is a static method that could go in another class, but we'll leave it here for simplicity of grading. This method will user the `WordScanner` to read through the file and build the `SortedPair` appropriately. Note, because we are using generics, this is where you will need to do lowercasing and make any restrictions on length (i.e. words must be 2 or more characters).

Once you finish this, you should be able to use the provided main method to run your program. We've also provided some text files in the assignment directory (which you can view using "ls" in the `Terminal`). When giving the full path of a filename, the filename will start at the very root '/' and then have the nested directories from there. An eccentricity with java is that you must have a double backslashes between directories when giving a file location in a String.

Experiment with different test files, but when you finally submit your program, make sure that you change it back to the original file that was there before ("/common/cs/cs062/assignments/assignment1/ctest.txt", but with double backslashes).

1.3 Submission

Follow the directions on the course web page for submitting. You should submit your `BetterInteger.java` and `SortedPairs.java` files. Be sure that your code is clear, formatted properly and commented appropriately (using Javadoc... see the first assignment for details on what's expected for comments).

1.4 Grading

You will be graded based on the following criteria:

criterion	points
functionality/correctness	12
appropriate comments (including JavaDoc)	3
appropriate use of generics	2
style and formatting	2
submitted correctly	1

NOTE: Code that does not compile will not be accepted! Make sure that your code compiles before submitting it.

2 The WordScanner class

The text here is intended to substitute for comments in the code.

A natural idea would be to extend the `Scanner` class. Unfortunately, the class is declared `final`, meaning that it cannot be extended. The alternative is to "wrap" the `Scanner` class in a new class.

```
protected Scanner theScanner;
```

The `Scanner` class has six constructors. We choose the four most useful-looking ones to "shadow" in `WordScanner`.

```
public WordScanner(File file) {
    try {
        theScanner = new Scanner(file);
    } catch (IOException e) {
        Assert.fail("File not found");
    }
}

public WordScanner(InputStream str) {
    theScanner = new Scanner(str);
}

public WordScanner(String name) {
    theScanner = new Scanner(name);
}

public WordScanner(Readable src) {
```

```
        theScanner = new Scanner(src);
    }
```

The first constructor is probably the one you will use. Create a `File` from a `String` with `new File(filename)` and use it as the argument to the `WordScanner` constructor.

The `Scanner` class uses powerful pattern-matching facilities from the `Pattern` class. We only need simple patterns here. The expression `[a-zA-Z]` matches one lower- or upper-case letter. The asterisk after it means "0 or more occurrences." Hence, `wordPattern` refers to an object that will match sequences of consecutive letters, which we think of as "words."

```
    protected static final Pattern wordPattern =
        Pattern.compile("[a-zA-Z]*");
```

The caret in `[^a-zA-Z]` means "anything but." So `separatorPattern` refers to an object that will match the non-letter characters between words.

```
    protected static final Pattern separatorPattern =
        Pattern.compile("[^a-zA-Z]*");
```

The patterns are arguments to the `Scanner` methods `skip` and `findInLine`. Those two methods do all the real work for us. The only public methods in `WordScanner` are `hasNext` and `next`. They are designed to be easily used in a `while` loop.

```
    protected void skipToNextWord() {
        try {
            theScanner.skip(separatorPattern);
        } catch (NoSuchElementException e) {
            // do nothing
        }
    }

    public String next() {
        skipToNextWord();
        return theScanner.findInLine(wordPattern);
    }

    public boolean hasNext() {
        try {
            skipToNextWord();
            return theScanner.hasNext();
```

```
        } catch (IllegalStateException e) {
            return false;
        }
    }
```

We do not rely on the `WordScanner` to shift letters to lower-case or to filter out the one-letter words; those tasks must be done elsewhere.