DUE: Thursday November 11, 2004, 11:59 p.m.

Introduction

The remaining homework assignments for this class will focus on building a compiler step by step. The compiler will take source files in a simple language that we will define later called FM (for flip movie) and will output Postscript files that describe the pages of a flip movie. This project was originally created by Doug Johnson.

You should comment your code (both public and private functions, if any) using javadoc style comments and run the javadoc utility to generate the documentation for your code.

CompilerIO

For this assignment, you will design, implement and test a Java class CompilerIO that will handle input and output for the compiler.

The idea behind this class is that the compiler will use a single CompilerIO object to manage input and output, which will shield the rest of the compiler from the details of file handling. The input and output files are ordinary text files.

Your CompilerIO class should meet the following requirements. There is also a javadoc page describing the class included in the homework download.

CompilerIO constructors

```
CompilerIO(String srcName, String objX) throws IOException
CompilerIO(String srcName, String objName, String objX) throws IOException
```

When the compiler starts, it will create an instance of the CompilerIO class. The CompilerIO constructor opens the input and output files and stores references to them for later use.

CompilerIO is required to have two constructors: CompilerIO(String srcName, String objX) and CompilerIO(String srcName, String objName, String objX). If an exception occurs while opening the files, the constructors should throw the exception and let the caller handle it.

The parameters for the two-argument constructor are the name of the input file and the extension for the object file (the output file). The constructor creates the output file name by taking the input file name (say test.fm) and replacing the extension with the given object extension objX (eg, giving test.ps). Don't make assumptions about the extension in the input file name. It might be ".fm", but it might also be ".txt" or something else. (Implementation hint: class String includes methods that locate the first or last occurrence of a character or substring in a String.) If the input file name has no extension, then a "." and the given extension are just added to the input file name to create the output file name. Once having created the output file name, the constructor opens both files to create a BufferedReader object (for the input) and a PrintWriter object (for the output) and stores references in appropriate instance variables.

The parameters for the three-argument constructor are the names of the input and output files and the extension for the output file. If the extension is not null, then the given extension is added to the output file name after a period. If the extension is null, then the output file name is used exactly as given. The constructor opens both files. The input file name is used to create a BufferedReader object, and the output file name is used to create a PrintWriter object. References to these objects are stored in appropriate instance variables.

Temporary buffer management

```
void openBuffer(), void closeBuffer()
```

In addition to writing to the output file, there will be times when we want to emit generated code to a temporary buffer so that we can later copy it to the output file several times.

Your class should implement openBuffer() and closeBuffer(). The openBuffer method creates a new empty temporary PrintWriter for buffering and sets it to be the target of all the emit methods (described next). closeBuffer() closes the temporary PrintWriter that we've been using to buffer the output and switches back to the output file as the target of all the emit methods. This method must be called before using emitBuffer to copy the contents of the temporary buffer to the output file.

Note that there at least two ways to implement this temporary buffer. You could use temporary files (see File.createTempFile(...)) or you could use a StringWriter. Either approach is okay. Potential exceptions may differ between the implementations.

Writing to the output file

```
void emit(String s)
void emitWithPrefix(String s)
void emitBuffer()
void emitFile(String fileName)
```

This group manages the process of sending generated object code to the output file. In our case this will be Postscript code. There are several methods defined. Note that each call to an emit function below should write a newline after it emits whatever is requested – this is easy to do if you use PrintWriter.println().

emit(String s) writes the given String to the currently active PrintWriter (either the output file or the buffer).

emitWithPrefix(String s) writes the defined prefix out then follows that with the given String on the same line. The prefix is presumably the comment delimiter for the target object language. This lets us write tracing information to the object file without causing execution errors. Also, when source lines are echoed while being read, they are preceded by the prefix. See below for more discussion of this prefix and the methods to control its use.

emitBuffer() writes the existing contents of the temporary buffer to the output file. This can only be called when the buffer is closed. This method will return silently without doing anything if the buffer is the currently active output PrintWriter.

emitFile(String fileName) copies an entire file to the currently active output Writer. This is a direct copy and nothing is done to the file on the way out, ie, no prefix is applied.

Reading from the input file

```
String readSrcLine() throws IOException
```

The scanner will use the CompilerIO object's readSrcLine() method to read the source program text. readSrcLine() should throw an exception if an error occurs while reading. This method reads one line of input from the input file and returns a String containing the contents of the line (not including any line-termination characters) or null if the end of the stream has been reached.

In addition to the generated code, we will want to include in the output file the original source code as comments to make the output easier to understand. So, besides just reading the input file, readSrcLine() should be able to automatically print each input line to the output file as it is read, if requested by the user with `setEchoing`() (see below).

Comment control

```
void setEchoing(boolean b), boolean getEchoing()
void setEchoPrefix(String s), String getEchoPrefix()
```

Instance variables (eg, `echoing` and `echoPrefix`) are used to control whether the input lines are printed to the output file, and how. If `echoing` is true, then each input line should be written to the output file at the time it is read, with the `echoPrefix` string concatenated to the front of it. The idea behind `echoPrefix` is that we can set it to something appropriate so the echoed source program lines copied to the output file are treated as comments in the generated program. You should provide methods to get and set both of these properties as described above.

Cleanup

```
void closeAll();
```

Close the input, output, and temporary streams.

Testing

You should write a separate test driver class named CompilerIOTest that exercises your CompilerIO class and shows that it works correctly. CompilerIOTest should have a main() method that runs all of the tests that you have written. A main method is not required for the class CompilerIO. You will probably want to create some sample input files to aid in this testing; use a ".txt" suffix on these files to make it easy to view them and to turn them in online.

Turn in

You should turn in 1.) CompilerIO.java, 2.) CompilerIOTest.java, 3.) any test input files that are needed by your CompilerIOTest, and 4.) the documentation page CompilerIO.html generated by javadoc. The link for this turnin is on the class calendar page. *You are not required to turn in hardcopy in class.*