CSE 114 Homework 3

Submission Information

Homework 3 is due by 11:59 PM on Friday, December 8. Submit your work (the `.java` source code files ***ONLY***, not the compiled .class files!) through the "Homework 3" link on Blackboard. You may submit an unlimited number of times; we will only grade the last/latest submission attempt, but be sure to attach **all** of your files to each submission attempt. **Be sure to include your name and Stony Brook ID number in a comment at the beginning of each file that you submit.**

General Description

Web pages are (usually) simple plain text documents written using a special notation called HTML (the HyperText Markup Language). HTML is a *markup language*; it consists of "tags" that are used to tell a Web browser like Chrome or Firefox how to display text in a document (Web page). For example, the `<b>` tag is used to indicate that certain text should be displayed in **boldface**.

HTML is far from the only markup language in common use. LaTeX is a markup language that is used for typesetting scientific and mathematical documents; its specialty is dealing with complicated mathematical formulas that use a wide range of letters and symbols. Markdown (developed by John Gruber and described at http://daringfireball.net/projects/markdown/basics) is another simple markup language that simplifies the task of writing HTML documents. Using Markdown, you can write a specially-formatted plain text document and then automatically convert it to HTML, with the appropriate HTML tags inserted where they belong.

Many business environments require programs that act as *filters*. These programs are required to translate data from one format to another (for compatibility between different programs, for example). For this assignment, you will write a simple filter program that converts a text file containing text formatted using Markdown into a basic HTML document (note that this assignment uses a weird hybrid of HTML 4 and HTML5 tags, so its result probably wouldn't pass a formal validation; that's all right for the time being).

A Brief Introduction to Markdown

For this assignment, we will only work with a limited subset of Markdown; we will not attempt to write a complete Markdown-to-HTML translator. The elements of Markdown that we will use are:

1. **Paragraphs**

   In Markdown, two or more consecutive blank lines start a new paragraph. This means that you need to insert a single `<p>` HTML tag in the output where you see two or more blank lines in a row.

2. **Emphasis**

   Markdown uses single asterisks at the beginning and end of a word or phrase to indicate a need for emphasis (usually either italics or boldface, at the Web browser's discretion).

   For example, the Markdown text

   ```
   Here is some *emphasized text.*
   ```

would be translated into the HTML sequence

```
Here is some <em>emphasized text.</em>
```

Strong emphasis uses double asterisks at the beginning and end, and replaces them with `<strong>` tags in HTML:

```
I **really** want to emphasize this point.
```

becomes

```
I <strong>really</strong> want to emphasize this point.
```

3. **Hyperlinks**

Markdown uses a combination of square brackets and parentheses to define links to other Web pages. The general syntax is as follows:

[*text for link*](*URL to link to*)

and is replaced by the equivalent HTML

text for link

For example,

```
See the [CSE 114 home page](http://www.cs.sunysb.edu/~cse114) for details.
```

becomes

```
See the <a href="http://www.cs.sunysb.edu/~cse114">CSE 114 home page</a>
for details.
```

There are no spaces between the closing square bracket and the opening parenthesis. This makes links easier to detect: if a block of text begins with a square bracket, we just keep reading until we reach the closing square bracket, and then read until we find the first closing parenthesis to find the end of the link (this is easier than it sounds).

4. **Images**

Markdown represents images much like hyperlinks. Images also use the “square bracket, parenthesis” format, except they begin with an exclamation point:

```
![alternate text](/path/to/image.jpg “Title text”)
```

becomes

```
<img src="/path/to/image.jpg" alt="alternate text" title="Title text">
```

For example,

```
![circles and squares](images/mysketch.jpg "My first sketch")
```

would become (in HTML):

```
<img src="images/mysketch.jpg" alt="circles and squares" title="My first
sketch">
```

5. **(Simplified) Code Tags**

   If Markdown text is enclosed in backtick characters (`), the text should be placed into HTML `<code>` tags, which will display it in a monospaced font appropriate for source code:

   ```
   The `String` class is critical for this assignment.
   ```

   would become

   ```
   The <code>String</code> class is critical for this assignment.
   ```

6. **Bulleted Lists**

   Markdown uses plus signs to represent a bulleted (unordered) list. Every line that starts with a plus sign forms a new item in the list. Don't forget to insert the `<ul>` and `</ul>` tags at the beginning and end of the list, so that it displays properly!

   ```
   + Larry
   + Moe
   + Curly
   + Shemp
   ```

   would become

   ```
   <ul>
   <li>Larry</li>
   <li>Moe</li>
   <li>Curly</li>
   <li>Shemp</li>
   </ul>
   ```

   For the purposes of this assignment, we will assume that list items do not contain any other formatting (emphasis, hyperlinks, etc.).

Instructions

Implement a small Java program that prompts the user to enter the name of a (plain text) input file and a name for a (plain text) output file. Your program should read the (Markdown-formatted) contents of the input file, one line at a time, and generate an appropriately-named output file that contains an HTML version of the input file's contents.

There are multiple ways to approach this task; we recommend implementing the following helper methods along the way:

1. A `translateEmphasis()` method. This method takes a single `String` argument representing the text that should be placed in HTML `<em>` tags. It returns a new `String` consisting of the argument wrapped in opening and closing `<em>` tags. For example, a call to

   ```
   translateEmphasis("blah blah");
   ```

   would return the `String`

   ```
   "<em>blah blah</em>"
   ```

2. A `translateStrongEmphasis()` method. This method works just like `translateEmphasis()`, except it adds HTML `<strong>` tags instead of `<em>` tags.

3. A `translateHyperlink()` method. This method takes two `String` arguments (some link text and a URL) and returns a `String` that contains a properly formatted HTML hyperlink according to the description given previously. For example, a call to

   ```
   translateHyperlink("my link", "http://www.cs.stonybrook.edu");
   ```

   would return the `String`

   ```
   "<a href=\"http://www.cs.stonybrook.edu\">my link</a>"
   ```

   (note the backslashes before the internal quotation marks, as well as the added angle brackets)

4. A `translateImage()` method. This method takes three `String` arguments (the alternate text, the image path, and the image title text) and returns a `String` that contains a properly formatted HTML image tag according to the instructions above. For example, a call to

   ```
   translateImage("picture of a dog", "images/dog.jpg", "Fido");
   ```

   would return the `String`

   ```
   "<img src=\"images/dog.jpg\" alt=\"picture of a dog\" title=\"Fido\">"
   ```

   (note the extra backslashes that are used to escape the internal quotation marks)

5. A `translateCode()` method. This method takes a single `String` argument, representing the text to be displayed in "code" style. It returns a `String` that consists of the input wrapped in opening and

closing HTML <code> tags. For example,

```
translateCode("System.out.println()");
```

should return the `String`

```
"<code>System.out.println()</code>"
```

6. A `translateListItem()` method, which takes a `String` argument representing some text. This method returns a new `String` consisting of its input wrapped in HTML list item (`<li>`) tags.

7. Your program should ultimately use a `Scanner` to process the input text line by line (you may assume that Markdown tags for emphasis, etc., do not span multiple lines). Be sure to insert HTML `<p>` tags to mark new paragraphs where appropriate.

   For each non-empty line of input, your general process will be the same: search the line for the different types of Markdown tags, rewriting the `String` as necessary. When the line has been checked for every type of Markdown tag, either append it to a "results" `String` or print it directly to the output file via a `PrintWriter` object.

   In order to handle bulleted lists correctly, you will need to use an additional variable (we suggest a `boolean`) that represents whether you are currently processing a list or not. If you encounter a list item and are *not* currently processing a list, be sure to generate an opening list tag (`<ul>`) before you translate the first list item (be sure to update your "are we actively processing a list?" variable as well!). Likewise, if you are currently processing a list and encounter a line that is *not* a list item, treat that as a sign that the current list has ended; generate a closing list tag (`</ul>`) and update your list-status variable.

   You may assume that list items always fit on a single line. You may also assume that list items are always contiguous (i.e. there are no blank lines between the items that make up a list). Finally, you can assume that lists will never be nested within one another.

8. Your output file's contents ***MUST*** begin with the following standardized HTML "boilerplate":

   ```
   <!DOCTYPE html>
   <html>
   <head>
   <title>Results of Markdown Translation</title>
   </head>
   <body>
   ```

   and ***MUST*** end with the following closing tags:

   ```
   </body>
   </html>
   ```

   with the translated contents of the input file inserted between the `<body>` tags.