CSE 143: Computer Programming II — Autumn 2016

HW2: HTMLManager (due Thursday, October 13, 2016 11:30pm)

This assignment focuses on using `Stack` and `Queue` collections. Turn in the following files using the link on the course website:

- `HTMLManager.java` – A class that manages the source code for a website.
- `HTMLManagerTest.java` – A client class that tests your `HTMLManager`.

You will need the support files `HTMLTag.java`, `HTMLTagType.java`, `HTMLParser.java`, and `HTMLMain.java` from the resources button on the course website; place these in the same folder as your program or project. If you are using EZclipse, the files will be automatically downloaded for you. You should not modify the provided files. The code you submit must work properly with the unmodified versions.

Although this assignment relates to websites and HTML, you will not need to know how to write HTML to complete the assignment.

What is HTML?

Web pages are written in a language called "Hypertext Markup Language", or HTML. An HTML file consists of text surrounded by markings called tags. Tags give information to the text, such as formatting (bold, italic, etc.) or layout (paragraph, table, list). Some tags specify comments or information about the document (header, title, document type).

A tag consists of a named element between less-than < and greater-than > symbols. For example, the tag for making text bold uses the element **b** and is written as `<b>`.

Many tags apply to a range of text, in which case a pair of tags is used:

- An opening tag (indicating the start of the range), which is written as: `<name>`
- A closing tag (indicating the end of the range), which is written as: `</name>`

So to make some text bold on a page, one would surround the text with opening and closing **b** tags:

Code: `<b>like this</b>`
Result: **like this**

Tags can be nested to combine effects. For example:

Code: `<b><i>bold italic</i></b>`
Result: ***like this***

Some tags, such as the `br` tag (which inserts a line break) or the `img` (which inserts an image), do not cover a range of text and are considered to be "self-closing." Self-closing tags do not need a closing tag; for a line break, only `<br>` is needed. Some web developers write self-closing tags with an optional / before the >, such as `<br />`.

Some tags can have attributes. For example, the tag `<img src="cat.jpg">` specifies an image from the file `cat.jpg`. The element is `img`, and the rest of the text such as `src` are attributes. In this assignment, you will not have to worry about attributes. The parser will store them for you, and you can just ignore them entirely.

HTML Validation

One problem on the web is that many developers make mistakes in their HTML code. All tags that cover a range must eventually be closed, but some developers forget to close their tags. Also, whenever a tag is nested inside another tag, `<b><i>like this</i></b>`, the inner tag (**i** for italic, here) must be closed before the outer tag is closed. So the following tags are not valid HTML, because the `</i>` should appear first: `<b><i>this is invalid</b></i>`

Here is an example of a valid HTML file (<!– and –> are comment tags):

```
<!doctype HTML public "-//W3C//DTD HTML 4.01 Transitional//EN">
<!-- This is a comment -->
<HTML>
   <head>
      <title>Turtles are cool</title>
         <meta http-equiv="Content-Type" content="text/HTML">
         <link href="style.css" type="text/css" />
   </head>

   <body>
      <p>
         Turtles swim in the <a href="http://ocean.com/">ocean</a>.
      </p>
      <p>
         Some turtles are over 100 years old. Here is a picture of a turtle:
         <img src="images/turtle.jpg" width="100" height="100"/>
      </p>
   </body>
</HTML>
```

In this assignment you will write a class that stores the contents of an HTML page and is able to fix any invalid HTML. Your `HTMLManager` will use stacks and queues to figure out whether the tags match and fix any mistakes it finds. Instructor-provided code will read HTML pages from files and break them apart into tags for you; it's your job to store the tags and fix the tags if there is a mismatch.

Implementation Details

Make sure to use *interface types* wherever possible.

You should...

- Write one class called `HTMLManager.java` that can handle pages with all types of `HTMLTags` (including self-closing tags). Your class must have the constructors/methods listed in the next section.
- Make sure that calling the methods multiple times in any order always gets the correct results.
- When using the `Stack` and `Queue` classes, you may only use the methods discussed in lecture, namely: add/push, remove/pop, peek, isEmpty, size and toString.

Several of your methods will need to interact with `HTMLTag` objects, which are described later in the spec.

HTMLManager

`HTMLManager` should have the following constructor:

`public HTMLManager(Queue<HTMLTag> page)`
The constructor takes in a Queue of HTMLTags that make up an HTML page. If the queue passed is null, the constructor throws an `IllegalArgumentException`. An empty queue (size 0) is allowed. You should **store a Queue as a field** to manage your tags, however, the constructor **should not directly store a reference to the given Queue** to avoid exposing internal state to the client.

Make sure to have a Queue as a field!

It should also implement the following methods:

`public void` **`add`**`(HTMLTag tag)`

Appends the given `HTMLTag` to the end of the queue of `HTMLTags` being managed. If the `HTMLTag` passed in is null, the method should throw an `IllegalArgumentException`.

`public void` **`removeAll`**`(HTMLTag tag)`

Removes all occurrences of the given `HTMLTag` from the queue of `HTMLTags` being managed. If the `HTMLTag` passed in is null, the method should throw an `IllegalArgumentException`.

`public List<HTMLTag>` **`getTags`**`()`

Returns a list of `HTMLTags` being managed. To avoid exposing internal state to your client, make sure to make a *copy* rather than just returning a reference to any internal field.

`public void` **`fixHTML`**`()`

This method will try to fix the page's HTML if there are any missing or extra tags. The algorithm that you will use is a simplified version of what Chrome, Firefox, and other browsers do when they try to display a broken webpage. This method should build up a correct version of the `HTMLTags` being managed, and update the instance to store the corrected version of the HTML when it is finished. To fix the HTML you will analyze the tags stored in the `HTMLManager` using a `Stack`.
The basic idea of the algorithm is to process the page tag by tag. For each tag, you will check to see if it has a matching tag later in the page in the correct place. Since self-closing tags don't have to match anything, whenever you see one, you can simply add it directly to the correct version of the tags. For opening tags, we assume that the writer of the HTML page intended to actually include the tag; so, like with the self- closing tag, we add it to the result. However, we need to keep track of if we have found its match; so, it should also be added to a `Stack`. If we find a closing tag, we must figure out if it is in the right place or not. In particular:

- If the opening tag at the top of the stack matches the closing tag we found, then it matches, and you should update the state according. (Hint: You probably want to edit the stack and the result.)
- If the opening tag at the top of the stack *does not match* the closing tag we found, then the writer of the HTML page made a mistake. To fix the mistake, you should add a new closing tag that matches the opening one at the top of the stack (so that it remains balanced). You should keep on adding closing tags to your result as long as the opening tag on top of the stack does not match. If you close all unclosed open tags on the stack, then the closing tag we found doesn't match any open tags and should be discarded.
- If, at any point, you find a closing tag that has no matching open tag, just discard it.

At the end of processing all the tags, if you have any left over tags that were never closed, you should close them. Note that every tag in the page that was ever opened *must* at some point be closed!
For one example, the html "`<b>hello</i>`" is invalid, and the algorithm would fix it to be "`<b>hello</b>`", because it would add in a new matching tag for the unclosed open **b** tag, and discard the extra closing **i** tag since there is no open **i** tag.
If the algorithm were given the invalid HTML "`<b><i>this is invalid</b></i>`", it would fix it to be "`<b><i>this is invalid</i></b>`".

fixHTML Example

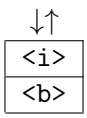
input: [, ,
, ,] output: []

First, we deal with the tag at the beginning. Since it's an opening tag, we *add it to our output* and *to our stack*, because we need to find its matching tag later:

input: [,
, ,] stack: output: []

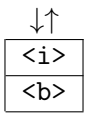
The next tag is another opening tag; so, we do the same idea again:

input: [
, ,] stack: output: [,]



The next tag is a *self-closing* tag; so, we don't need to save it to find the mate. We only add it to our output:

input: [,] stack: output: [, ,
]



Now, we hit a closing tag. Since the opening tag at the top of the stack *does not match* the closing tag we found, the HTML writer made an error. We fix this error by inserting a matching tag into our output and popping the open tag off the stack:

input: [,] stack: output: [, ,
,]

This time, we hit a closing tag that *does match* the one we found. So, we pop it off the stack and append the tag we found:

input: [] stack: output: [, ,
, ,]

Finally, since there are no tags left on the stack, the closing tag we just hit must be an extra. So, we just discard it.

input: [] stack: output: [, ,
, ,]

Since the input and stack are both empty, we're done!

HTMLTag

An `HTMLTag` object corresponds to an HTML tag such as `<p>` or `</table>`. An `HTMLTag` can either be an opening tag, a closing tag, or a self-closing tag. You don't ever need to construct `HTMLTag` objects in your code, but you will process them. For examples of how to construct the `HTMLTag` objects, see the starter testing code. `HTMLTag` objects know how to handle HTML content, but you do not have to worry about the content at all. The reason we use `HTMLTag` objects instead of just storing the tags as strings is that the class provides extra functionality that makes processing `HTMLTags` easier.

In particular, it provides the following methods:

`isOpening()`	Returns true if **this** HTML tag is an "opening" (starting) tag.
`isClosing()`	Returns true if **this** HTML tag is an "closing" (closing) tag.
`isSelfClosing()`	Returns true if **this** HTML tag is an "self-closing" tag.
`matches(`**other**`)`	Returns true if the given **other** tag matches (has the same tag type as) **this** tag; e.g., `<a>` and `</a>`.
`getMatching()`	Returns **a new tag** that matches **this** tag but has opposite type (e.g. if this is `<a>`, then this method will return `</a>` and vice versa)
`equals(`**other**`)`	Returns true if the given **other** tag equals **this** tag. The **other** tag is equal if the internal element is the same as **this** tag's element and has the same opening or closing state; e.g. `<a>` and `<a>`.
`toString()`	Returns a string representation of **this** HTML tag, such as "`</a>`".

These methods will be useful throughout writing your program. Note that you *do not* ever have to read the `HTMLTag` (or `HTMLParser`) code; though, you may if you want to.

Your Test Case (`HTMLManagerTest.java`)

In addition to `HTMLManager.java`, you will create tests to verify your class's `removeAll` method. Create a file `HTMLManagerTest.java` to thoroughly test the `removeAll` method (a starter file is available on the website). Your tests should call `removeAll` at least three times, removing three different elements. You should compare the tags stored in your `HTMLManager` after each call with the expected Queue of tags.

Your testing program must produce at least three lines of output indicating whether each of your tests passed. You will be evaluated partly on how complete your tests are (whether they try every possible input and state configuration).

Development Strategy

The best way to write code is in *stages*. If you attempt to write everything at once, it will be significantly more difficult to debug, because any bugs are likely not isolated to a single place. For this assignment we will provide you with a development strategy. We will also provide you with some correct output that you can check using the Output Comparison Tool on the website.

We suggest that you develop the program in the follow four stages:

(1) In this stage, we want to decide what field(s) belong in the `HTMLManager` class. These field(s) should be initialized in the constructor.

(2) In this stage, we want to make sure that the constructor is working by printing out the page we constructed. The best way to do this is to write `getTags()` and print out the returned List.

(3) In this stage, we want to implement the method that edit the page we currently have. You should implement `add(HTMLTag tag)` and `removeAll(HTMLTag tag)`.

(4) Finally, we want to write the difficult method: `fixHTML()`. You can test that things are working by using the Output Comparison Tool on the course website.

null

The value `null` is a special value that indicates the lack of an object; a reference that does not refer to any object. When a given method's spec says, "if foo is null, do X," it means that you should test:

`if (foo == null) { X }`.

In particular, a `null` queue is not the same as an empty queue; a `null` string is not the same as the empty string, "" ; and a `null` `HTMLTag` is not the same as an `HTMLTag` with a `null` element (which will not occur anyway, since `HTMLTag` throws an exception if you try to create one with a null element).

Style Guidelines and Grading

Unless otherwise specified, your solution should use only material covered so far. Part of your grade will come from appropriately using the allowed methods of `Stacks` and `Queues`.

Avoid Redundancy

Create "helper" method(s) to capture repeated code. As long as all extra methods you create are `private` (so outside code cannot call them), you can have additional methods in your class beyond those specified here. If you find that multiple methods in your class do similar things, you should create helper method(s) to capture the common code.

Data Fields

Properly encapsulate your objects by making data your fields `private`. Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used in one place. Fields should always be initialized inside a constructor or method, never at declaration.

Java Style Guidelines

Appropriately use control structures like loops and if/else statements. Avoid redundancy using techniques such as methods, loops, and factoring common code out of if/else statements. Properly use indentation, good variable names, and types. Do not have any lines of code longer than 80 characters.

Commenting

You should comment your code with a heading at the top of your class with your name, section, and a description of the overall program. All method headers should be commented as well as all complex sections of code. Comments should explain each method's behavior, parameters, return values, and assumptions made by your code, as appropriate. The `ArrayIntList` class from lecture provides a good example of the kind of documentation we expect you to include.

Make sure you describe complex methods inside methods

You do not have to use the pre/post format, but you must include the equivalent information– including the type of exception thrown if a precondition is violated. Write descriptive comments that explain error cases, and details of the behavior that would be important to the client (What does it do if the tag is not found? Where does it add the tag? What happens if it is null? Etc.). Your comments should be written in your own words and not taken verbatim from this document.