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.

You will need the support files HTMLTag.java, HTMLTagType.java, HTMLParser.java, and HTMLMain.java, and cse143.util from the resources button on the course website; place these in the same folder as your program or project. 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: bold italic
```

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):

```html
<!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

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.
- Use the provided Stack and Queue classes from `cse143.util`.
- Make sure that calling the methods multiple times in any order always gets the correct results.
**HTMLManager**

Because you will be using cse143.util, you will have to write your imports in a certain order to avoid any ambiguities. Your imports should be as follows:

```java
1 import java.util.List;
2 import java.util.ArrayList;
3 import cse143.util.*;
```

HTMLManager should have the following constructor:

```java
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.

It should also implement the following methods:

```java
public void add(int index, HTMLTag tag)
```

This method adds the given tag (which is guaranteed to not be null) to the HTML page at the given index. If index is invalid (larger than the number of tags or less than 0), the method should throw an IllegalArgumentException.

For example, if the current HTML page were `[<div>, <a>, </a>, </div>]` and we called `add(2, <i>)`, the result would be: `[<div>, <a>, <i>, </a>, </div>]`

Note that it is **valid** to insert at the end of the page.

```java
public void remove(int index)
```

This method removes the tag at the given index from the HTML page. If index is invalid (larger than the number of tags or less than 0), the method should throw an IllegalArgumentException.

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

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

```java
public String toString()
```

This method returns a String representing the HTMLTags. This method should just concatenate the tags together and return the result.

Continued on next page
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 make the instance store the corrected version of the HTML when it is finished. To fix the HTML you will analyze the tags in 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 result. 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 accordingly. (Hint: You probably want to edit the stack and the output.)
- 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 fixing mistakes until the next tags match.
- If, at any point, you find a closing tag that has no matching open tag, just discard it.

Note that every tag in the page that was ever opened must at some point be closed! For example, the html "<b>hello</i>" is invalid, and the algorithm would fix it to be "<b>hello</b>", because there is no open i tag. Also, if the algorithm were given the invalid html "<b><i>this is invalid</i></b>", it would fix it to be "<b><i>this is invalid</i></b>".

**HTMLETag**

An HTMLETag object corresponds to an HTML tag such as <p> or </table>. An HTMLETag can either be an opening tag, a closing tag, or a self-closing tag. An HTML comment is a special kind of self-closing tag. You don’t ever need to construct HTMLETag objects in your code, but you will process them. The reason we use HTMLETag objects instead of just storing the tags as strings is that the class provides extra functionality that makes processing HTMLETags easier. In particular, it provides the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isOpening()</td>
<td>Returns true if this HTML tag is an “opening” (starting) tag.</td>
</tr>
<tr>
<td>isClosing()</td>
<td>Returns true if this HTML tag is an “closing” (closing) tag.</td>
</tr>
<tr>
<td>isSelfClosing()</td>
<td>Returns true if this HTML tag is an “self-closing” tag.</td>
</tr>
<tr>
<td>isComment()</td>
<td>Returns true if this HTML tag is a comment.</td>
</tr>
<tr>
<td>matches(other)</td>
<td>Returns true if the given other tag matches (has the same tag type as) this tag; e.g., &lt;a&gt; and &lt;/a&gt;.</td>
</tr>
<tr>
<td>getMatching()</td>
<td>Returns a new tag that matches this tag but has opposite type (e.g. if this is &lt;a&gt;, then this method will return &lt;/a&gt; and vice versa)</td>
</tr>
<tr>
<td>toString()</td>
<td>Returns a string representation of this HTML tag, such as “&lt;/a&gt;”.</td>
</tr>
</tbody>
</table>

These methods will be useful throughout writing your program. Note that you do not ever have to read the HTMLETag (or HTMLParser) code; though, you may if you want to.
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 will not provide you with any testing code this time, and we strongly recommend you write some of your own.

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

1. In this stage, we want to decide what fields belong in the HTMLManager class. These fields should be initialized in the constructor. So, you’ll want to begin by creating the fields and 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 toString().

3. In this stage, we want to implement the method that edit and reveal the page we currently have. You should implement add(), remove(), and getTags() here.

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
Some students have trouble understanding the directions related to null on this assignment. 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 cse143.util 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. 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.