CSE 143, Winter 2011
Programming Assignment #3: HTML Validator (40 points)

Due Thursday, January 27, 2010, 11:30 PM

This program focuses on using `Stack` and `Queue` collections. Turn in files named `HtmlValidator.java`, `mytest.html`, and (optionally) `HtmlValidatorTest.java` from the Homework section of the course web site. You will need `HtmlTag.java` and `ValidatorMain.java` from the web site; place them in the same folder as your program.

*Though this assignment relates to web pages and HTML, you do **not** need to know how to write HTML to complete it.*

Background Information About 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 and a *closing* tag indicating the end of the range. A closing tag has a `/` slash after its < symbol, such as `</b>`. So to make some text bold on a page, one would put the text to be bold between opening and closing `b` tags, `<b>`**`like this`**`</b>`. Tags can be nested to combine effects, `<b><i>`***`bold italic`***`</i></b>`.

Some tags, such as the `br` tag for inserting a line break or `img` for inserting 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 a tag of `<br>` is needed. Some web developers write self-closing tags with an optional `/` before the >, such as `<br />`.

The distinction between a tag and an element can be confusing. A tag is a complete token surrounded by `<>` brackets, which could be either an opening or closing tag, such as `<title>` or `</head>`. An element is the text inside the tag, such as `title` or `head`. Some tags have **attributes**, which are additional information in the tag that comes after the element. 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 we will ignore attributes and focus just on elements and tags.

If you're curious, there are tutorials such as http://www.w3schools.com/html/ and http://www.cs.washington.edu/190m/.

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>`**

Below is an example of a valid HTML file, with its tags in bold. A tag of `<!-- ... -->` is a comment.

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

  <body>
    <p>My name is Marty Stepp.  I teach at <a href="http://washington.edu/">UW</a>.</p>
    <p>I have been a teacher here since 2003.  Here is a picture of my cat:
       <img src="images/kitteh.jpg" width="100" height="100">
    </p>
  </body>
</html>
```

In this assignment you will write a class that examines HTML to figure out whether it represents "valid" sequences of tags. Your **validator** will use **stacks and queues** to figure out whether the tags match. Instructor-provided code will read HTML pages from files and break them apart into tags for you; it's your job to see whether the tags match correctly.

Implementation Details:

You will write a class named `HtmlValidator`. You must use Java's `Stack` and `Queue` from `java.util`. Your class must have the constructors/methods below. It must be possible to call the methods multiple times in any order and get the correct results each time. Several methods interact with `HtmlTag` objects, described later. Unless otherwise specified, you may not create auxiliary data structures (such as arrays, lists, stacks, queues) to help you solve any method below.

`public` **`HtmlValidator`**`()`
`public` **`HtmlValidator`**`(Queue<HtmlTag> tags)`

Your class should have **two constructors**. The first should initialize your validator to store an empty queue of HTML tags. The second should initialize your validator to store a given queue of HTML tags. For example, the queue for the page shown previously would contain the tags below. Further tags can be added later if the client calls `addTag`.

```
front [<!doctype>, <!-- -->, <html>, <head>, <title>, </title>, <meta>, <link>,
      </head>, <body>, <p>, <a>, </a>, </p>, <p>, <img>, </p>, </body>, </html>] back
```

If the queue passed is `null`, you should throw an `IllegalArgumentException`. An empty queue (size 0) is allowed. The constructors are allowed to construct a queue to store your validator's tags if necessary.

`public void` **`addTag`**`(HtmlTag tag)`

In this method you should add the given tag to the end of your validator's queue.
If the tag passed is `null`, you should throw an `IllegalArgumentException`.

`public Queue<HtmlTag>` **`getTags`**`()`

In this method you should return your validator's queue of HTML tags. The queue contains all tags that were passed to the constructor (if any) in their proper order; it should also reflect any changes made such as adding tags with `addTag` or removing tags with `removeAll`.

`public void` **`removeAll`**`(String element)`

In this method you should remove from your validator's queue any tags that match the given element. For example, if your validator is constructed using the tags from the page shown previously and `removeAll("p")` were called on it, your queue would be modified to contain the following tags. Notice that all `<p>` and `</p>` tags have been removed:

```
front [<!doctype>, <!-- -->, <html>, <head>, <title>, </title>, <meta>, <link>,
      </head>, <body>, <a>, </a>, <img>, </body>, </html>] back
```

If the element passed does not exactly match any tags (such as an empty string), your queue should not be modified. You may not use any auxiliary collections such as extra stacks or queues, though you can create simple variables.

If the element passed is `null`, you should throw an `IllegalArgumentException`.

`public void` **`validate`**`()`

In this method you should print an indented text representation of the HTML tags in your queue. Display each tag on its own line. Every opening tag that requires a closing tag increases the level of indentation of following tags by four spaces until its closing tag is reached. The output for the HTML file on the first page would be: *(continued on next page)*

```
<!doctype>
<!-- -->
<html>
    <head>
        <title>
        </title>
        <meta>
        <link>
    </head>
    <body>
        <p>
            <a>
            </a>
        </p>
        <p>
            <img>
        </p>
    </body>
</html>
```

To generate the output, analyze your queue of tags with a `Stack`. The basic idea of the algorithm is that when you see an opening tag that is not self-closing, you should push it onto a stack and increase your indentation. When you see a closing tag, you should pop the top element from the stack and decrease your indentation. You may use a single temporary `Stack` (in addition to your validator's own queue of tags) to help you compute the result. You may not use any other collections, arrays, etc., though you can create as many simple variables as you like.

Error Handling:

Your `validate` method should print error messages if you encounter either of the following conditions in the HTML:

- A closing tag that does not match the most recently opened tag (or if there are no open tags at that point).
- Reaching the end of the HTML input with any tags still open that were not properly closed.

For example, the following HTML is valid:

```
<p><b>bold text <i>bold and italic text</i> just bold again</b> <br/> more </p>
```

But the following HTML is *not* valid, because the `</b>` appears before the `</i>`:

```
<p><b> bold text <i>bold and italic text</b> just italic</i> neither</p>
```

The following HTML is also *not* valid, because the `<html>` tag is never closed:

```
<html><body> <b><i>bold italic</i></b> normal text</body>
```

Suppose the previous short HTML file were modified to add several errors, as follows: an added unwanted `</!doctype>` tag, a deleted `</title>` tag, an added second `</head>` tag, and a deleted `</body>` tag:

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

  <body>
    <p>My name is Marty Stepp.  I teach at
       <a href="http://www.washington.edu/">UW</a>.</p>
    <p>Here is a picture of my cat:
       <img src="images/kitteh.jpg" width="100" height="100"> </p>
</html>
```

The resulting output for this invalid file should be the following:

```
<!doctype>
ERROR unexpected tag: </!doctype>
<!-- -->
<html>
    <head>
        <title>
            <meta>
            <link>
ERROR unexpected tag: </head>
ERROR unexpected tag: </head>
            <body>
                <p>
                    <a>
                    </a>
                </p>
                <p>
                    <img>
                </p>
ERROR unexpected tag: </html>
ERROR unclosed tag: <body>
ERROR unclosed tag: <title>
ERROR unclosed tag: <head>
ERROR unclosed tag: <html>
```

The reason that there are two error messages for `</head>` are because neither `</head>` tag seen matches the most recently opened tag at the time, which is `<title>`. The four unclosed tags at the end represent the fact that those four tags didn't have a closing tag in the right place (or, in some cases, no closing tag at all).

Because of the simplicity of our algorithm, a single mistake in the HTML can result in multiple error messages. Near the end of the file is a `</html>` tag, but this is not expected because `body`, `title`, and `head` were never closed. So the algorithm prints many errors, such as saying that the `html` tag is unclosed, though the underlying problem is that the `body` tag was never closed. Also notice that an unexpected closing tag does not change the indentation level of the output.

Your **revised validation algorithm**: Examine each tag from the queue, and if it is an opening tag that requires a closing tag, push it onto a stack. If it is a closing tag, compare it to the tag on top of the stack. If the two tags match, pop the top tag of the stack. If they don't match, it is an error. Any tags remaining on the stack at the end are errors.

Provided Files:

• `HtmlTag.java`:	Objects that represent HTML tags for you to process.
• `ValidatorMain.java`:	A testing program to run your `HtmlValidator` code and display the output.

An **`HtmlTag`** object corresponds to an HTML tag such as `<p>` or `</table>`. You don't ever need to construct `HtmlTag` objects in your code, but you will process them from your queue. Each object has the following methods:

`public String getElement()`
Returns this HTML tag's element name, such as `"table"`.

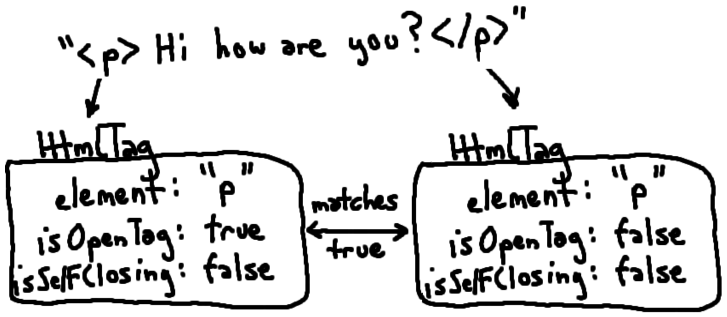
`public boolean isOpenTag()`
Returns `true` if this tag is an opening tag, such as `<p>` or `<html>`.

`public boolean isSelfClosing()`
Returns `true` if this element does *not* require a closing tag, which is the case for elements such as `br` and `img`.

`public boolean matches(HtmlTag other)`
Returns `true` if this tag and the given tag have the same element but opposite types, such as `<body>` and `</body>`.

`public String toString()`
Returns a string representation of this HTML tag, such as `"<p>"` or `"</table>"`.

Some students have confusion about the difference between an `HtmlTag` object and a `String`. An `HtmlTag` is related to a `String` in that it stores an element as a `String` inside of it; and in that a tag has a natural representation as a `String` by surrounding its element with < and > brackets. But the `HtmlTag` object is more useful than a bare `String` in that it has the methods above. So you can ask it if it is an opening tag, whether it matches another tag, etc. without needing to manually trim away < and > characters or other drudgery.



It may seem like a chore to have to figure out how `HtmlTag` objects work, when `String`s are more familiar. But part of becoming a mature software developer is becoming comfortable with pre-written code that is provided to you by others, and using it to solve part of a larger overall task.

Your Test Case (`mytest.html`, and optional `HtmlValidatorTest.java`):

In addition to `HtmlValidator.java`, you will also create a test case of your own to help verify your validator. Create a file **`mytest.html`** with any contents you like, so long as it has at least 10 total HTML tags. This can be a file you create from scratch, or you can go to an existing web page and save it to a file. The point is to make you think about testing.

In addition, you can earn **+1 extra late day** if you can write and submit an *optional* JUnit test program for your validator named **`HtmlValidatorTest.java`**. At a minimum, your program should contain 2 testing methods, where each constructs a validator with a given queue of tags and asserts aspects of its behavior. (See the provided `SimpleTest` as an example to follow.) You may share your test cases with other students on the message board if you like.

Development Strategy and Hints:

We suggest the following development strategy for solving this program:

1. Create the class and declare every method. Leave every method's body blank; if necessary, return a "dummy" value like `null` or `0`. Get your code to run in the `ValidatorMain` program, though the output will be incorrect.
2. Implement the bodies of the constructors, `getTags`, and `add` methods. Verify them in the testing program.
3. Write the `removeAll` method.
4. Write an initial version of `validate` that assumes the page is valid and does not worry about errors. Get the overall algorithm, output, and indentation working for valid HTML. Use the algorithm described previously.
5. Add the `validate` code that looks for errors and prints appropriate error messages, as described previously.

You can test the output of your validator by running it on various inputs in the `ValidatorMain` client. You can also use our Output Comparison Tool to see that your outputs match what is expected.

`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).

The `validate` method is the toughest. It can be hard to handle all of the errors properly and to get the indentation just right. Remember that the indentation increases when an opening tag is seen, and the indentation decreases when a valid expected closing tag is seen. Those should be the only two times you ever need to adjust your indentation.

Some students also have trouble with `removeAll`. Recall that there are common stack/queue bugs related to looping over a stack or queue whose contents and/or size are being changed during the loop. See the textbook appendix on stacks and queues posted on the class web site for ideas about how to avoid these bugs.

Style Guidelines and Grading:

Part of your grade will come from appropriately utilizing stacks and queues. You may only use their methods shown in lecture and section; it is forbidden to use them in ways that are not stack/queue-like. For example, you may not call any `Stack` methods that accept index parameters. You may not examine a stack/queue using a "for each" loop or iterator. Do not make unnecessary or redundant extra passes over a queue when the answer could be computed with fewer passes. Declare queues as variables of type `Queue`, not variables of type `LinkedList` (on the left side of the equals sign).

Redundancy is always a major grading focus; avoid redundancy and repeated logic as much as possible in your code.

Properly encapsulate your objects by making fields `private`. Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used within a single method. Initialize fields in constructors only.

Follow good general style guidelines such as: appropriately using control structures like loops and `if/else` statements; avoiding redundancy using techniques such as methods, loops, and `if/else` factoring; properly using indentation, good variable names, and proper types; and not having any lines of code longer than 100 characters.

Comment descriptively at the top of your class, each method, and on complex sections of your code. Comments should explain each method's behavior, parameters, return, pre/post-conditions, and any exceptions thrown. 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.

For reference, our solution is around 100 lines long (54 "substantive"), though you do not have to match this exactly.