Programming Assignment 3

DOM Tree

In this assignment you will implement an HTML Document Object Model (DOM) Tree.
Browsers create DOM trees for web pages, to facilitate the manipulation of HTML entities by client-side code such as Javascript.

Worth 100 points (10% of your course grade)

Posted Fri, Mar 8

Due Fri, Mar 29, 11:00 PM (NO GRACE PERIOD)

Extended deadline (with ONE time extension pass): Mon, Apr 1, 11:00 PM (NO GRACE PERIOD)

You get ONE free extension pass for assignments during the semester, no questions asked. There will be a total of 5 assignments this semester, and you may use this one free extension pass for any of the 5 assignments.

A separate Sakai assignment will be opened for extensions AFTER the deadline for the regular submission has passed. The extension will be 3 days (72 hours). If/when you choose to use the one-time extension, you don't need to ask for permission - just drop your submission in the extension assignment. And you can do this even if you dropped something in the regular submission, and later on decided to use the extension - in this case, only the extension submission will be graded, the regular submission will be ignored.

- You will work **individually** on this assignment. Read the DCS Academic Integrity Policy for Programming Assignments - you are responsible for this. In particular, note that **"All Violations of the Academic Integrity Policy will be reported by the instructor to the appropriate Dean".**
- **IMPORTANT - READ THE FOLLOWING CAREFULLY!!!**

  Assignments emailed to the instructor or TAs will be ignored--they will NOT be accepted for grading.
  We will only grade submissions in Sakai.

  If your program does not compile, you will not get any credit.

  Most compilation errors occur for two reasons:

  1. You are programming outside Eclipse, and you delete the "package" statement at the top of the file. If you do this, you are changing the program structure, and it will not compile when we test it.
  2. You make some last minute changes, and submit without compiling.

  To avoid these issues, (a) START EARLY, and give yourself plenty of time to work through the assignment, and (b) Submit a version well before the deadline so there is at least something in Sakai for us to grade. And you can keep submitting later

versions (up to 10) - we will accept the LATEST version.

Summary

You will write an application to build a Document Object Model (DOM) for a given HTML file, and process it with a set of given input commands that will transform the tree.

Document Object Model

The Document Object Model (DOM) is a *platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page.* (This is quoted from the W3C Document Object Model specification web page.)

When you access a web page, the browser builds a DOM tree structure for the HTML content in the page.

Consider the following HTML example:

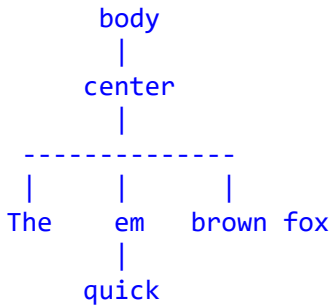
```
<html>
<body>
<center>The <em>quick</em> brown fox</center>
</body>
</html>
```

which is rendered by a browser as:

The *quick* brown fox

The DOM tree for this HTML content would look like this:

```
        html
         |
```



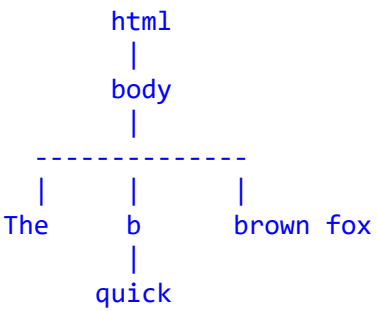
Note: There is a space character after "The", and before "brown" so the corresponding nodes of the tree actually store `"The "` and `" brown fox"`.

You can see the DOM tree for any HTML document in the browser. In Chrome, go to View --> Developer --> Inspect Elements. In Firefox, go to Tools -> Web Developer -> Inspector.

Open the sample.html file for the above example in your browser, and look at its DOM. You will notice that under the root html node, in addition to the body node, there is also a head node (with nothing under it), which corresponds to the head tag. The head tag is not used in the example above, and is automatically supplied by the browser. The HTML documents in this assignment will not have a head tag. You will also notice the trailing space character included in "The " and the leading space character included in " brown fox".

Web developers can build a "dynamic" HTML page by embedding, say, Javascript code in the page, that can manipulate the DOM tree. For example, such code can show or hide individual columns in an HTML table upon user request.

In the example above, you could have Javascript code within the page make calls to the DOM tree, to replace the `em` tag with the `b` tag, and remove the `center` tag, for the following result:



The browser keeps track, and renders the tree onto the web page like this:

The **quick** brown fox

Bring up try.html in your browser. This page has javascript code embedded in HTML, that replaces `em` with `b` or vice versa when you click a button. It does this by changing the DOM tree representation of the page.

You can view the web page's HTML source using the keyboard shortcut option-command-u (Mac) or ctrl-u (Windows)

Restricted HTML for Assignment

For this assignment, you will work with the following restricted set of HTML *tags*:

`html`	top level
`body`	second level
`p`	paragraph
`em`	emphasis (italics)
`b`	boldface
`table`	table
`tr`	row (within `table`)
`td`	column (within `tr`)
`ol`	ordered (numbered) list
`ul`	unnumbered list
`li`	list item (within `ol` or `ul`)

Moreover, the format of the HTML document itself (supplied through an input file) will be restricted to the following:

- Every tag will consist of the tag name (such as `p`) surrounded by < and >

  There will not be anything else in a tag, not even spaces.

  So `<p>` is good, but `< p>` is not (space before p), and `<p id="007">` is not (tag has an attribute named id)

- Every tag will appear on a line by itself - there will be nothing else on that line
- Every HTML file will mandatorily have the `html` and `body` tags
- The HTML will be well-formed, in that every tag will have a closing counterpart. For instance, every tag construct of the form `<b>` will be closed by `</b>`
- If there is a `table` tag in the document, there will only be one such tag. (In other words, there won't be more than one `table` tag, if at all there is one.)
- A table can have any number of rows (`tr` tags nested under `table`), and a row can have any number of columns (`td` tags nested under `tr`), but all rows will have the same number of columns.
- Nesting of the *same type* of tags is only permitted with `ol` and `ul` tags. That is, an `ol` tag may have another `ol` tag nested within it, and a `ul` tag may have another `ul` nested within it. But you may NOT have a `b` tag within a `b` tag, etc.
- There will not be any tags not defined in the above list

Examples

Following are some sample HTML pages. Click on a link to see the page, and view the page source to see the underlying HTML:

ex1.html	`p`, `em`, and `table` tags
ex2.html	`ol` and `ul` tags
ex3.html	Nested `ol` and `ul` tags

DOM Tree Structure

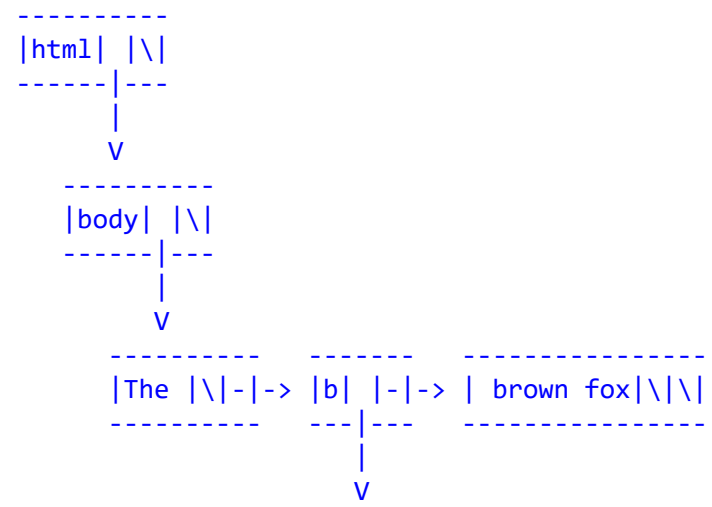
Since the nodes in the DOM tree have varying numbers of children, the structure is built using linked lists in which each node has three fields: the **tag** (which can be an HTML tag or plain text), the **first child** (which is null if the tag is plain text), and the **sibling**, which is a pointer to the next sibling.

As an example, consider the following input HTML:

```
<html>
<body>
The
<b>
quick
</b>
 brown fox
</body>
</html>
```

(The third line is actually `"The "`, i.e. trailing blank space, and the "brown fox" line has a leading space before "brown".)

The DOM tree implementation for this HTML would look like this:



```
-----------
|quick|\|\|
-----------
```

Note: Tree nodes containing tags do NOT include angle brackets with the tags.
So, if a node stores the tag `b`, it stores the string "b", NOT the string "".
Also, closing tags (e.g. "/em") are NOT be stored in the tree.
If your tree violates these requirements, it will be considered incorrect!

Implementation

Download the attached `domtree_project.zip` file to your computer. DO NOT unzip it. Instead, follow the instructions on the Eclipse page under the section "Importing a Zipped Project into Eclipse" to get the entire project into your Eclipse workspace.

You will see a project called `DOM Tree` with the following classes:

- `structures.TagNode`
- `structures.Tree`
- `structures.Stack`
- `apps.DOM`

There are also a number of sample test files directly under the project folder (see the **Examples** section that follows.)

Note: You are not required to use the included `Stack` class. But if you do use a stack, you MUST use this class (not your own).

You will implement the following methods in the `Tree` class:

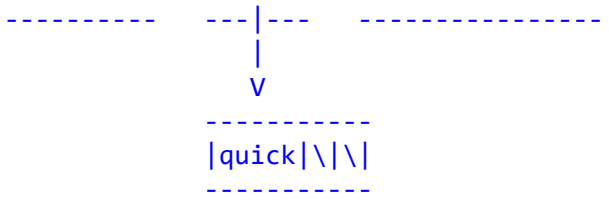
- **(20 pts)** `build`: Builds the tree from an input HTML file
- **(15 pts)** `replaceTag`: Transforms the tree by replacing *all* occurrences of a tag with another (e.g. replace `b` with `em`, or `ol` with `ul`.

  Only sensible replacements will be asked, you need not do any checking. (For instance, `ol` cannot be replaced with `em`, so you will not be asked to do this.)
- **(30 pts)** `removeTag`: Transforms the tree by deleting *all* occurrences of a tag. There are two categories of tags that can be deleted:
  - Category 1: `p`, `em` and `b` tags: All occurrences of such a tag are deleted from the tree.

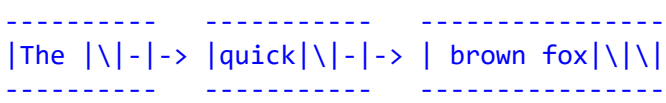
    Note: Text nodes should NOT be merged if the removal of a tag results in an unbroken string of text.

    For instance, removing tag `b` in this:

```
----------   -------   ----------------
|The |\|-|-> |b| |-|-> | brown fox|\|\|
```



should give this:



In other words, the three nodes above should NOT be merged into a single text node.

- Category 2: `ol` or `ul` tags: All occurrences of such a tag are deleted from the tree. In addition, all the `li` tags immediately under the removed tag are converted to `p` tags.

- **(15 pts)** `boldRow`: Transforms the tree by boldfacing a given row of a table. **This is done by bold facing the content of every column in the given row.** (It should NOT add a single bold face tag for the entire row.) Also see the header comment for this method in the code.

  You may assume that the row will not already have been boldfaced. Also note that the first row is row number 1 (not zero), the second row is row number 2, etc.

- **(20 pts)** `addTag`: Transforms the tree by adding a tag around all instances of a *taggable word* (case INsensitive match). The added tag can only be either `em` or `b`, nothing else.

  A *taggable word* is a sequence of alphabetic letters (nothing else) that is not a part of another word, and may have one of the following punctuations AS THE LAST CHARACTER: period '.', comma ',', question mark '?', exclamation point '!', colon ':' and semicolon ';'.

  The taggable word may appear within a longer piece of text and may occur multiple times in the containing text.

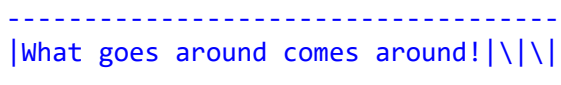
  In the following example:

  ```
  What goes around comes around!
  ```

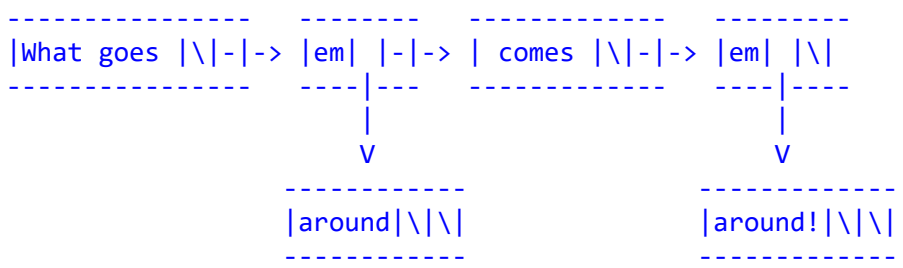
  the word "around" is tagged with `em`:

  ```
  What goes <em>around</em> comes <em>around!</em>
  ```

  Before tagging:

After tagging:



Notice that the punctuation goes with the word when tagged. Also note that the blank space between "goes" and "around" will now trail "goes" in the DOM tree node, and the spaces between "around" and "comes" in "around comes around!" will now lead and trail "comes" in the DOM tree node.

As another example, suppose you were asked to add a bold tag around "cow" in the following cases:

- `cow`
- `cow!`
- `cow?`
- `cow.`
- `cow;`
- `cow:`
- `COW`
- `cOw`

This would be the result after tagging, for each (the last two are case insensitive matches):

- `<b>cow</b>`
- `<b>cow!</b>`
- `<b>cow?</b>`
- `<b>cow.</b>`
- `<b>cow;</b>`
- `<b>cow:</b>`
- `<b>COW</b>`
- `<b>cOw</b>`

But you would NOT tag "cow" in these words:

- `cows`, `cowabunga`
  In both cases, `cow` is not a word by itself, but part of a longer word
- `?cow`
  Although `?` is one of the acceptable punctuations, it does not meet the requirement of being the last character
- `cow?!!`
  Only one punctuation character at the end is acceptable

- `cow's`
  Apostrophe is not a letter

Examples of Transformations

Here are examples of applying transformations to some HTML pages, and the resulting HTML (look inside the source files as well):

Input	Transformation	Result
ex1.html	replace `em` with `b`	ex1tr1.html
ex1.html	delete `em`	ex1tr2.html
ex2.html	delete `ol`	ex2tr1.html
ex3.html	delete `ol`	ex3tr1.html
ex1.html	boldface 2	ex1tr3.html
ex2.html	add `em` around item	ex2tr2.html

Rules of implementation

- You may NOT modify any of the files except `Tree.java` in **ANY** way.
- You may NOT make **ANY** modifications to `Tree.java` EXCEPT:
  - Write in the bodies of the methods you are asked to implement
  - Add private helper methods as needed

Running the Program

The `DOM` class in the `apps` package is the application driver. When you run it, it asks for an input html file name. It creates a scanner for this file, which it then passes to the `Tree` constructor. Then it calls the `Tree` build method to build the DOM tree.

Following this, it goes into a loop, and presents a menu of actions from which you can pick whether you print the tree structure, or the HTML generated from the tree, or do one of the transformations.

Here's a sample run with the tree built out of the ex1.html input file:

```
Enter HTML file name => ex1.html

Choose action: (p)rint Tree, (h)tml, (r)eplace tag, (b)oldface row, (d)elete tag, (a)dd tag, or (q)uit? => p

     html
      |----body
            |----p
```

```
            |----A line of non-tagged text.
      |----p
            |----A
            |----em
                  |----new
            |----paragraph.
      |----table
            |----tr
                  |----td
                        |----em
                              |----R1C1
                  |----td
                        |----R1C2
            |----tr
                  |----td
                        |----R2C1
                  |----td
                        |----R2C2
```

NOTE: The tree structure printed via the `print()` method as above is the test for correctness.

```
Choose action: (p)rint Tree, (h)tml, (r)eplace tag, (b)oldface row, (d)elete tag, (a)dd tag, or (q)uit? => h

<html>
<body>
<p>
A line of non-tagged text.
</p>
<p>
A
<em>
new
</em>
paragraph.
</p>
<table>
<tr>
<td>
<em>
R1C1
</em>
```

```
</td>
<td>
R1C2
</td>
</tr>
<tr>
<td>
R2C1
</td>
<td>
R2C2
</td>
</tr>
</table>
</body>
</html>

Choose action: (p)rint Tree, (h)tml, (r)eplace tag, (b)oldface row, (d)elete tag, (a)dd tag, or (q)uit? => q
```

CAUTION: The HTML generated off the tree using the `getHTML()` method as above does NOT by itself guarantee correctness of the tree (the same HTML can be generated off different trees.) But after verifying with the `print()`ed tree structure, you can use the HTML to see the result in a browser page.

Submission

Submit your **Tree.java** file ONLY.

(If you submit a .class or .zip, or any other file, you will get no credit even if you have a working version on your computer. We will only grade whatever is in Sakai.)

Grading

All input files will be correctly formatted, so you don't have to do any error checking for input format.

Every transformation that is applied will be on a legitimate tree structure, where the transformation should work without failure.

Your `build` method will be graded first. After that, all other methods will be graded using our correct `build` implementation.

Each of the non-build methods will be graded using a tree built from scratch. This means every test case is treated separately, so that it does not depend on any previous test case's transformation of the tree.

Since every method you will implement results in building or modifying the DOM tree structure, we will grade by examing the tree structure that results at the end of each of your methods. Starting with `root` instance field, the grading process will traverse your tree structure and compare it with the expected structure. (As stated earlier in the **Running the Program** section, we will NOT be looking at the HTML printout of the tree.)

To check the tree structure as you go, you may either use the `print()` method, or you may want to use the Eclipse debugger to navigate through the tree nodes. There is comprehensive information on how to use the debugger in Eclipse under Help -> Help Contents -> Java development user guide -> Tasks -> Running and Debugging, and Help -> Help Contents -> Java development user guide -> References -> Views -> Variables View. Essentially, you need to know how to set breakpoints and examine variable values when the program stops at a breakpoint. (And of course, there are YouTube videos on using the debugger in Eclipse.)