Project 4: JavaWord Back-End

1 Task

1.1 Request

Your team is developing a simple word processor which will later integrate into a larger system. You are asked to write the first version of the back end, used to store, save, and load that data into memory. As random access is less important than the flow of document text between sections and paragraphs, you choose to store the data using *linked lists*.

1.2 Basics

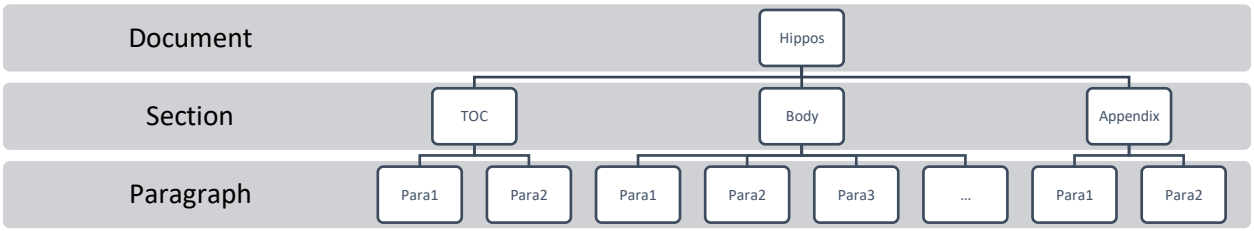
Paragraphs are made up of text plus a paragraph style. Styles we currently support include: left, center, and right alignment; bulleted and numbered lists; and heading levels one through four[1]. Client code must be able to assign and retrieve both properties.

Sections are made up of paragraphs of text. Client code must be able to retrieve a paragraph count, plus retrieve a specific paragraph by its position. Client code should also be able to add a paragraph (at the end of the section, by default), add paragraphs at a specific position in the list, and remove paragraphs at a specific position. You'll also need to support move operations including moving a specific paragraph "up" in the document and moving it "down" in the document (always within the same section).

Documents (which you can think of as a "document workspace," if that helps) are made up of a name (which can be a simple file name or a full path), plus a series of sections. Client code should have similar methods for accessing Sections as they do for Paragraphs. In addition, documents must support normal file operations including: open (via deserialization), close, new, save (via serialization), and save to HTML. When saved, documents should have the extension wpd (word-processing document).

Because of design constraints, the team has decided that only one document can be open at a time; they aren't ready to deal with multi-doc scenarios like what happens if two documents are operating on the same file at the same time. Make it so that only one document can be open within the application. Also manage state carefully, e.g., "open" is only allowed from a closed state, "close" from an open state.

Conceptual Levels[2]



[1] These are mutually exclusive options, e.g., you can't have a right heading 1 style. All heading levels are left aligned.
[2] Don't interpret this is as an object diagram; it's just a visual so you can understand how documents are formed.

1.3 Sample Main Code

Here's a sample of Main code that will create a simple document. Your code will look slightly different based on your implementation.

```
...
myDoc.newDoc("Hippos");
Section firstSect = new Section();

firstSect.addParagraph(
    new Paragraph("Pygmy Hippos of Africa", Paragraph.ParaStyle.Heading_1));
firstSect.addParagraph(
    new Paragraph("While the hippopotamus exists in various places in Africa..."));
firstSect.addParagraph(new Paragraph("Hippo facts:"));
String bulletedText = "";
bulletedText += "The name Hippopotamus comes from the Ancient Greek 'river horse'. \n";
bulletedText += "Hippos secrete an oily red substance; they do not sweat blood. \n";
bulletedText += "An adult Hippo resurfaces every 3 to 5 minutes to breathe.\n";
bulletedText += "They are only territorial while in the water.";
firstSect.addParagraph(
    new Paragraph(bulletedText, Paragraph.ParaStyle.List_Bulleted));
firstSect.addParagraph(
    new Paragraph("I hope you have enjoyed our foray into the world of the pygmy hippo..."));

myDoc.addSection(firstSect);
myDoc.save();
myDoc.saveToHtml();
```

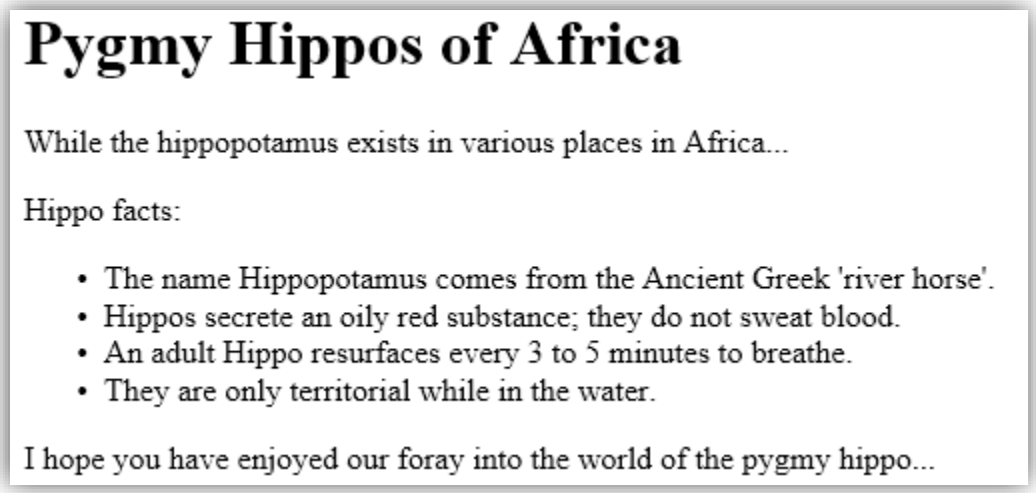
(project handout continues…)

1.4 Save to HTML

When a document is saved to HTML, it should produce valid output that includes basic tags to make the document look reasonable when viewed in a browser. Sections are not demarcated, but simply appear one after another.

```
<!DOCTYPE html>
<html>
   <body>
      <h1>
         Pygmy Hippos of Africa
      </h1>
      <p style="text-align:left">
         While the hippopotamus exists in various places in Africa...
      </p>
      <p style="text-align:left">
         Hippo facts:
      </p>
      <ul>
         <li>The name Hippopotamus comes from the Ancient Greek 'river horse'.</li>
         <li>Hippos secrete an oily red substance; they do not sweat blood.</li>
         <li>An adult Hippo resurfaces every 3 to 5 minutes to breathe.</li>
         <li>They are only territorial while in the water.</li>
      </ul>
      <p style="text-align:left;">
         I hope you have enjoyed our foray into the world of the pygmy hippo...
      </p>
   </body>
</html>
```

Output, when viewed in the browser:



1.4.1 Notes

You aren't expected to support special characters in this version (e.g., quotation marks must be written in HTML as `&quot;`). For now, assume the document will contain only simple text that won't require special treatment.[3]

1.4.2 Extra Credit

1.4.3 HTML Indentation

For the base version, all tags can be written at the left margin. For extra credit, produce nicer tabbed alignment (as shown above), with indentation visually indicating tag nesting. Ideally this will be an *extensible* solution, considering that there may be more outer or inner tags in the future.

1.4.4 HTML Lists

For the base version, you can write out bulleted and numbered lists as regular left-aligned paragraphs. For extra credit, write out bulleted lists using the <ul> tag (unordered list), and numbered lists using <ol>. You'll need to break up the paragraph at new line characters (see sample document creation above), so you can write out each individual list item (<li>). Hint: String's split method is your friend.

2 Design Documentation

Before you code, create appropriate design documentation and obtain feedback. Update designs per feedback, then use them during the rest of the development process and submit them as part of your project. This should include a UML Class Diagram and a UML Object Diagram.

3 Code Implementation

3.1 Additional Building Blocks You'll Need

- Doubly linked lists (your own implementation), with each node having a next and prev reference
- The typical "front" reference, but also an "end" reference that points to the end of the list
- Generics, as you'll need more than one list, here
- Static variables and static code blocks
- Singleton design pattern

3.2 Requirements

Here are requirements for your coding project:

- Provide a main class to show off what your code can do. Write a small amount of code that reaches across a wide cross-section of objects in the model and show its power.
- Do not expose client code to nodes; in fact, make it so this isn't even *possible*. The underlying architecture might change in the future. Plus, we want our lists to maintain full control over themselves; exposing nodes would client code make a mess of things, and we can't have that.

3.3 Additional Requirements

Important: there will be additional requirements that we'll determine together after you have some time to ponder potential design approaches to this project. These are a required part of your design, even though they aren't written in this document.

[3] Here's a link listing special characters and how they need to be written in HTML.

3.4 Style

Follow the Course Style Guide. You'll lose points on every assignment if you fail to do so.

4 Testing

Create a JUnit test class for each production class. Ensure that each method and state is fully tested. This must include constructors, accessors, mutators, and preconditions. You do not need to test UI-heavy methods like toString and similar.

You may wish to create test files; if so, make sure they are in your project folder, so they'll be included with your submission. Name these files contextually, e.g., "TestFile-3Pars.txt" rather than "TestFile1.txt" as it'll be easier for us both to know the file's purpose. Put these in your project folder and include them with your submission.

5 Submitting Your Work

Follow the course's Submission Guide when submitting your project.

6 Hints

- Before coding this project, carefully consider design feedback and implement appropriate changes.
- There are plenty of subtleties required in manipulating linked lists; fall back to pencil and paper to ensure you know how to create and break links, and in what order to do them.
- Some linked-list methods can be made more efficient because you have a doubly linked list and an end pointer; *revisit each method* and consider what improvements might be made.
- Think carefully about return values from methods. If you can return information useful to client code, then do so, especially in cases where the code makes a request that you must gently refuse; a boolean return can let them know "thanks, but I couldn't do that for you."
- Think carefully about what to write to the serialized file; consider whether writing the entire document is the right approach, or whether writing its contents is better and easier.

(project handout continues…)

7 Grading Matrix and Points Values

Area	Value	Evaluation
Design docs	10%	Did you submit initial design documents, and were they in reasonable shape to begin coding? Did you submit final design documents, and were they a good representation of the final version of the project?
Classes created	20%	Were the non-list classes (plus main) successfully created? Were good coding skills evident in that work? Were requested node manipulation methods implemented correctly and successfully?
Doubly linked list	10%	Was a doubly linked list implemented, and were references properly maintained in all scenarios? Were list methods streamlined based on the availability of end and prev references?
Generics	10%	Were generics implemented successfully? Are there any warnings remaining, when scanned with -Xlint:unchecked?
Save/Open	10%	Were save and open correctly implemented? Can they, together, successfully "round trip" a document?
Save to HTML	15%	Are documents correctly saved to HTML? Can the document be viewed correctly in a browser, and does it look reasonable when viewed in an editor?
Singleton implementation	5%	Was the Singleton design pattern implemented correctly?
Style/internal documentation	10%	Did you use JavaDoc notation, and use it properly? Were other elements of style (including the Style Guide) followed?
JUnit tests	10%	Were test classes created for each production class? Were all possible methods tested? Was all state tested?
Extra credit: HTML indentation	2%	Was tag nesting shown via indentation, and is this framework extensible, should new tag levels be added later? Is the resulting HTML easy to read?
Extra credit: HTML lists	2%	Were bulleted and numbered lists written out properly to HTML, and do they look good in a browser?
Total	**104%**	

Code that does not compile or run won't be graded