Assignment 2: The Information Superhighway

Data Structures, CSCI-UA 102, Section 7
Instructor: Max Sklar

Build a text-based web browser for the early World Wide Web.

Goals

- Use stacks to implement back and forward on a web browser.
- Practice using an existing class or interface to build new functionality.
- Work with an in-application REPL interface.
- Learn about the early web, and the challenges involved in getting the first websites functional.

Story

The year is 1993, and you are part of a small team trying to understand and popularize the so-called World-Wide-Web. Many commentators dismiss the internet as a passing fad. You disagree. Some politicians and marketing gurus are calling it the "Information Superhighway". Will that stick? Probably not. For now, you believe that this is at the cutting edge of computing. Time to place your bets and start building!

Background

The World Wide Web was created by Sir Tim Berners-Lee in 1991 while working at CERN, the European Organization for Nuclear Research. His goal was to allow researchers to share information more easily over the Internet, and his invention revolutionized communication. The first website, hosted at CERN, was little more than a simple text page with links, but it laid the foundation for the web as we know it today.

Berners-Lee developed the first web browser/editor, called WorldWideWeb, on a NeXT computer. NeXT was founded by Steve Jobs after he was removed as CEO of Apple and before returning to Apple in 1997. This browser supported graphical content, but NeXT machines were not widely available. As a result, most early users of the web relied on text-based browsers like Lynx, which could only display text and navigate through links using the keyboard. The graphical potential of the web wouldn't become widespread until later, as most machines of that era were limited in their display capabilities.

In 1993, the introduction of Mosaic, the first widely adopted graphical web browser, transformed the web experience by allowing images to be displayed alongside text. Mosaic played a key role in popularizing the web, offering features such as inline images and a user-friendly interface that worked across different operating systems. Before Mosaic, web navigation was linear—users followed links from one page to the next without the convenience of forward or back buttons.

This assignment takes you back to the early days, where you will build a simple, text-based web browser similar to those used before Mosaic. While your browser won't have all the modern capabilities, it will help you understand the foundations of web technology and the challenges involved in early web navigation.

Instructions

Your job is to complete the implementation of the `Browser.java` class, which browses Berners-Lee's original website. The `Browser` class will browse web pages, navigate between pages, and handle user input through the provided `TextRenderer` object.

The `Browser` class already has some structure, but several important commands are missing, such as handling page navigation (`NEXT`, `PREV`, `BACK`, `FORWARD`), displaying links, and reloading pages. Your task is to fill in these missing commands, ensuring that the browser can navigate web pages properly and manage user input.

What You'll Be Working With

- `TextRenderer`: This class is responsible for rendering web pages and managing the links found on each page. It's already provided to you, and you'll interact with it through the `Browser` class to display content and navigate frames (displayable sections of a web page).
- **Stacks for Navigation**: The `Browser` class will use two sets of stacks—one for page navigation (back and forward) and another for navigating between frames. The class `ArrayStack` and interface `Stack` from the Goodrich book (slightly modified) are also provided.

You will complete the following tasks:

- Implement commands `NEXT`, `PREV`, `BACK`, `FORWARD`, `RELOAD`, `HOME`, and `LINKS`.
- Implement link following, which users will do by entering the number of the link they want to follow as the command.
- Manage browser history and current state using stacks for both page and frame navigation.
- Ensure the browser interacts correctly with the `TextRenderer` to display web content and links.

The skeleton for the `Browser.java` class has already been provided, along with the `TextRenderer.java` and `UrlInfo.java` classes, so your focus will be on filling in the core logic to handle user commands.

Frames

Older machines did not scroll; a fixed number of rows could appear on the monitor at any one time. Therefore, some websites might need to be shown over multiple *frames*. Each frame contains a maximum number of lines allowed by the particular system this Browser is running on. A single command line input is going to specify the number of lines we can render at a time (or the number of lines in a frame).

```
int numLinesAtATime = Integer.parseInt(args[0]);
```

Spinning up the Browser and UrlInfo

The `main` method starts by creating an instance of the Browser class with numLinesAtATime and a hard-coded home page.

```
String homepage = "info.cern.ch/hypertext/WWW/index.html";
String homepageUrlInfo = new UrlInfo(homepage, null)
Browser browser = new Browser(homepageUrlInfo, numLinesAtATime);
```

The UrlInfo class, provided in UrlInfo.java, contains two String fields: path and hash. This is necessary because some URLs will contain a string after the hash (#) symbol to indicate that navigation starts at a particular location in the middle of the page. For example, one of the links in History.html is People.html#BernersLee.

You won't need to worry about breaking apart this string. This is handled automatically in the static function navigate.

```
public static UrlInfo navigate(String currentPath, String link)
```

When a user wants to navigate to a new url, simply give it the path of the current URL, and the link as a String from the TextRenderer.links array, and it will return the new UrlInfo that the user wants to see.

Real-Eval-Print-Loop, or REPL

Once the Browser object is created, the process goes into a while loop, called a Read-Eval-Print-Loop or REPL. The REPL is a command line interface that reads user input, evaluates the command (including making any updates to internal objects), and prints a response. Then, we loop and a new command is read. You have seen the REPL pattern before in a variety of places, including your own terminal interface.

The only time the repl stops is when it receives the `EXIT` command. Then, System.exit(0) is called and the process ends. Although commands are listed in uppercase here, this system is not case sensitive.

Commands

The first two commands are already implemented for you.

- **HELP**: Print a list of all the commands and what they do.
- **EXIT**: Also QUIT, stop the process and exit the browser.

You need to implement these other REPL commands to get the browser to work.

- **HOME**: Return to the home page of the browser.
- **NEXT**: This user is telling the browser that they want the next frame (in modern terms: scroll down).
- **PREV**: This user is telling the browser that they want the previous frame (in modern terms: scroll up).
- **RELOAD**: Reload the current page from the beginning.
- **BACK**: Return to the previous page that I was on.
- **FORWARD**: Return to the next page in my forward page stack. This only works after going BACK one or more times
- **LINKS**: Print a list of links on the current page and their associated numbers.
- [**number**]: If the user types a number and it is within the appropriate range for the links array, the browser should navigate to the corresponding new page.

Using Stacks

There are 2 pairs of stacks (4 total) that you will need to maintain in order to get the Browser running correctly.

The first pair of stacks should be of type

```
Stack<UrlInfo>
```

and it should keep track of all the previous pages that you've navigated to. This is required to get the `BACK` command working. There is also a forward stack so that the `FORWARD` command can work as well.

When a user clicks on a link (by using the link number as a command), then the current page should be put onto the back stack, and the forward stack should be cleared.

When a user goes BACK, the current page should be pushed onto the forward stack, and the back stack should be popped to retrieve the page that the user is looking for.

When a user goes FORWARD, the current page should be pushed onto the back stack, and the forward stack should be popped.

Sometimes, it is not possible to go forward or back, or a link is broken. In that case, return a simple message like "No page to go back to." and simply do not carry out the operation.

The second set of stacks is for the frames, and these should be of type

`Stack<String>`

A long page will need to be displayed in multiple frames. In order to display the next frame, the user gives the command `NEXT`. When this happens, the current frame goes onto the frame back stack, and a new frame is retrieved, which in most cases means going to the TextRenderer (more on that below) and calling nextFrame() after ensuring that it has another frame.

When a user gives the command `PREV`, the current frame is pushed onto the forward frame stack, and the back frame stack is popped to retrieve the previous frame.

The forward frame can be used to retrieve frames that the Browser already has before going back to the renderer. So, when a user hits NEXT, the first thing to do is to check the forward frame stack. If that is non empty, pop it to get the new current frame. If it is empty, then go to the renderer. If renderer.hasNextFrame() is false, then there is no next frame and a short message like "End of page reached." should be returned and printed by the REPL.

TextRenderer.java

The TextRenderer takes website URL, loads the associated file (which is in an older version of HTML), and turns that file into strings that can be output using System.out.println().

The TextRenderer also keeps track of the links it sees in an array as it processes the web page. It stores a link to the current page in position 0, and it counts the links sequentially.

There are 5 public methods and fields in TextRenderer.java.

- `public TextRenderer(int rows)`

  The constructor takes in the number of rows in a frame, which is the number of lines that your computer will render in one setting. This is called numLinesAtATime in our Browser class.

- `public boolean newPage(UrlInfo urlInfo)`

  This tells the renderer that it is about to render a new page. If it successfully finds the new page, then it throws away all the data that it has about the old page and returns true. If it cannot find the new page, it returns false, and it stays with the existing page.

- `public boolean hasNextFrame()`

  Call this method to ask the renderer if there are any additional frames that need to be shown to the user on this page.

- `public boolean nextFrame()`

  Call this method to get the next frame of text from the renderer. This frame will be limited in size by the rows field, set in the constructor.

- `public List<String> links`

  This field contains an ArrayList of all of the links found on the current page so far.

Hints

- Here are the fields your instructor used in Browser.java:

  ```
  final private UrlInfo homepage;

  private Stack<UrlInfo> pageBackStack = new ArrayStack<>();
  private Stack<UrlInfo> pageForwardStack = new ArrayStack<>();
  private UrlInfo currentPage;

  private Stack<String> frameBackStack = new ArrayStack<>();
  private Stack<String> frameForwardStack = new ArrayStack<>();
  private String currentFrame;

  private TextRenderer renderer;
  ```

- The function showHelp() uses a StringBuffer to build a string. This is preferable to continuously appending to a string using something like

  ```
  s += "next line\n"
  ```

  because that operation copies the string each time, leading to an $O(n^2)$ running time. You may use the StringBuffer elsewhere, particularly for the `LINKS` command.

- A few of the commands are very easy to implement. Don't doubt your answer just because it was simple! The navigational ones tend to have more steps.

- There are a few steps involved in navigating to a new link. Once you've made sure that a given link number is valid, you can get the link url by going to the renderer.links array. From there, you can get the new UrlInfo by calling UrlInfo.navigate like so:

  ```
  UrlInfo urlInfo = UrlInfo.navigate(currentPage.path, renderer.links.get(linkNumber)
  ```

  The rest of it requires clearing out the frame stacks and handling the back and forward page stacks. Don't forget to push the current page (before following the link) on to the back page stack, and also to clear the forward page stack!

- A formatLink() method is provided for you in Browser.java, so that you can print the links sequentiall on the `LINKS` command.

- You can design the error messages in a way that makes sense to you.

Code Style

Your code style will be checked with the following directives in mind:

- Your code should be readable and understandable.
- You may select any reasonable style (i.e., for indentation or bracket placement), but it should be consistent.
- Please add comments in any part of the code where it is not obvious to the reader what the code is doing and why.
- Please use descriptive and efficient names for your classes, functions, and variables. Local counters that can be still be called something like i for index.

Submission

The entire submission should be a zip file containing all relevant files and folders and should be uploaded to Brightspace by the due date. **The zip file should be of the form [netid].zip where you insert your netid**. The zip file includes:

- Your code, which should include one or more Java files and may or may not be organized into packages. This includes code provided for the assignment.
- Your `readme.txt` file.

README

Included in the `readme.txt` file should be the following:

RUNBOOK: Information on how to compile and run your code in the command line.

TIME SPENT: An estimate of the time spent working on this project.

NOTES: Include the difficulties and roadblocks encountered while working on this project, including notable bugs that needed to be overcome. Are there any quirks in your solution that we should be aware of? Was the outcome what you expected?

RESOURCES AND ACKNOWLEDGEMENTS: Disclose the sources you consulted for help on this project. Possible human sources include: NYU Staff (the instructor, TA, tutor), other students, and anyone who helped you with the project. Sources that you consulted, including code that you found online, in the book, and output from LLMs. Describe the nature of your LLM usage.

There will never be a grading penalty for a complete and accurate accounting of resources and acknowledgments.