Assignment 2

Web Browser (Part 2)

Due: not later than Friday, Jan. 12, 2018, 11:59 p.m.

Description:

In this Assignment you will add additional features to the web browser you built in Assignment 1. This includes:

- Shortcut keys for the various menu items
- A History table indicating recently viewed pages
- Appropriate response from the address bar when the carriage return is entered
- View HTML/JavaScript/CSS
- Exception Handling, for web pages that contain non-standard HTML or JavaScript
- The ability to remove bookmarks
- JavaDoc documentation for all classes, methods and variables

Worth
12%
of your total mark

Assignment 2

Web Browser (Part 2)

I. Create a new project.

a. In Eclipse, create a new project called `Assignment2`, with a package named `assignment2` inside

b. Copy the package from Assignment 1 into the Assignment2 `src` folder, and refactor as necessary.

(You can, of course, take your Assignment 1 project folder and refactor it to Assignment 2 directly. But then you'll have nothing to fall back on if your code becomes hopelessly convoluted and refuses to work *at all*. So be smart: leave your Assignment 1 code intact in its original folder, and start Assignment 2 in a new, separate project folder.)

II. Add the following features to your existing web browser application, as described below

a. Add accelerator/shortcut keys to each menu/menu item, as follows.

First, to indicate which key is the shortcut key, place the underscore character directly before the character that will act as the shortcut. So, for example, to load a menu with Ctl+'F' used as the accelerator key, you would use

```
Menu mnuFile = new Menu("_File");
```

Note that you usually need to press the 'Alt' key first for this mnemonic character to appear in the menu. (This may vary with OS; if 'Alt' doesn't work, try the Control key.)

This only indicates to the user which key to press; it does not actually have any effect unless you assign the menu/menu item to the accelerator key itself. To do this, use the `setAccelerator()` method associated with each `MenuItem`. This works just like `setOnAction()`, but it takes a `KeyCombination` object (or alternately, one of its subclasses, such as a `KeyCodeCombination` object) rather than an `EventHandler` as its parameter.

For more information, see:

http://java-buddy.blogspot.ca/2012/02/javafx-20-set-accelerator.html

https://blog.idrsolutions.com/2014/04/tutorial-how-to-setup-key-combinations-in-javafx/

https://stackoverflow.com/questions/24499500/javafx-menu-first-letter-underline-decoration

b. Display the browser history in the right panel of the browser's right borderpane using the `WebEngine`'s `getHistory()` method. To do this, you'll first need to add a new 'History' menu item under Settings which, like the address bar, should toggle the History pane on and off.

The `getHistory()` method returns a WebHistory object. This object in turn has two methods of interest to us. First, a `getEntries()` method, which returns an `ObservableList` of type `<WebHistory.Entry>`. As with all `ObservableList` objects, you have the ability to `get()` any item in the list. This is not really so different from an `ObservableList` of Strings, except that `WebHistory.Entry` has methods like `getURL()`, `getTitle()`, and, of course, `toString()`, that are related to web-based objects.

Second, `getHistory()` has a `go()` method, which takes an integer, *n*, as its argument and causes the `WebEngine` to jump *n* pages forwards or backwards in the history list. Thus, for any `WebEngine we`, `we.getHistory().go(1)` loads the next page in the history (if there is one), and `we.getHistory().go(-1)` goes back to the previously-viewed web page.

This is important, since your application will need to display the history list by title, and, using 'backwards' and 'forwards' buttons at the bottom of the panel, and load up the currently-selected webpage from the History list into the WebEngine.

For this part of the application, we'll ignore selections from the history list; your code does not need to respond to mouse clicks on the history list and load up the corresponding page. Instead, assume that when the history page is loaded up, we jump to the *last* item in the list—the most-recently loaded page. Pages are then loaded as the forward and backward buttons are clicked, and a new page is thus selected from the history list.

Note that your code should *NOT* generate *any* exceptions in the process, i.e. your code should not attempt to access a history list item having an index less than 0 or greater than the number of `WebHistory.Entry`'s in the `ObservableList`.

This following web pages may provide useful to you in this part of the assignment:

https://stackoverflow.com/questions/18928333/how-to-program-back-and-forward-buttons-in-javafx-with-webview-and-webengine

http://www.chegg.com/homework-help/questions-and-answers/assignment-2-goal-lab-create-web-browser-javafx-giving-code-start-assignment-must-add-extr-q11537943

Note that `Platform.runLater()` allows your code to run separately from the main thread. Your application would still run without using `runLater()`, but it would 'lock up' the browser for several seconds until the new web page was loaded. `Platform.runLater()` offloads the processing to a separate thread, making execution appear seamless. (Multithreading is a subject you'll be seeing more of in your Level III and IV courses.)

Aside from this note, much of the information on these pages (and most of the other pages on this subject) contain far more information than you'll need to display the history list. For one thing, since you are *not* expected to handle mouse clicks on the items in the history list, you'll have no need to add a `Listener` to your code, amongst other things.

This code should not be difficult to implement, assuming you use the `go()` method responsibly.

Regarding the forward and back buttons: you can add a graphic if you wish. But a much easier way to add graphics to buttons is to load Unicode characters instead. For example, u23EE, u23F4, 23F5 and u23ED contain the standard symbols for first, previous, next and last—although you may wish to play around with the default font size.

c. In Assignment 1 you added an address bar that contained a textfield object that allowed the user to enter a URL and go to that location by clicking on the 'Go' button. At this point we wish to add two new features to the address bar.

   First, when the user presses the enter key (with the cursor in the textbox) we'd like to be able to use this event to trigger loading the new page. We can add an event handler that listens only to keypresses, and waits for the enter key to be pressed, as follows

```
txt.setOnKeyPressed(e -> {
   if (e.getCode() ==
               KeyCode.ENTER){
      //load URL to engine
    }
});
```

Numerous examples can be found on the web, e.g.

https://stackoverflow.com/questions/13880638/how-do-i-pick-up-the-enter-key-being-pressed-in-javafx2/13881850

https://tagmycode.com/snippet/1522/listen-for-enter-key-event-on-a-textfield#.WkSEUTdG1XI

etc. Your code must load the in the textfield URL to the web engine when the enter key is pressed.

The second feature you need to add is this (if it isn't already implemented): ideally, every time the web page changes in the browser, this should be reflected in the address bar. This *should* happen regardless of the source of the web page change. It should happen when the user enters a new URL in the textfield, certainly, but it should also happen when the user navigates through the history list, clicks on a bookmark, or selects a link from inside a web page.

Fortunately, the web engine allows you to detect changes in the current web page, and this new information can be loaded into the address bar. You'll need to add code to the WebPage class provided with Assignment 1. The following gives an indication of how to implement this feature:

https://stackoverflow.com/questions/32486758/detect-url-changes-in-javafx-webview

d. In addition to a History pane, you'll also need to add an option (under Settings) to display the HTML/JavaScript code for the current web page. As with History, the 'Display Code' option should be toggleable, alternately ON or OFF. The code for this process is convoluted, but fortunately we don't need to concern ourselves with the details. The steps are straightforward and always the same. A good source can be found at

http://www.java2s.com/Tutorials/Java/JavaFX/1500__JavaFX_WebEngine.htm

About 3/5 down this page you'll find the heading "Obtain the raw XML data as a string of text". As before, we're not interested in anything related to `addListener()`, since you've already loaded the WebView and WebEngine. But the code inside the try – catch block, including the declaration for transformerFactory and such, will generate a text file that you can display in the bottom pane of the border pane.

Note that you will need the try – catch block to ensure that the program works reliably.

e. On this note, it's worth adding that not all HTML is well-behaved. And even when it is, not all browsers are capable of handling all the features available in HTML, JavaScript and CSS. Therefore, an additional feature of your browser program is that it must capture (and ignore) the following exceptions

- MalformedURLException
- StringIndexOutOfBoundsException
- IllegalArgumentException

The former is typically thrown by the WebEngine whenever a bad URL is passed to it. Therefore your code should use a try – catch around the appropriate block of code where this could occur. (Note: if you have more than one such block of code, you're not practicing proper code reuse. All loading of the web page should take place at one location only.)

One way to test the address bar to a Malformed URL is to load the string into the URL class constructor, i.e.

```
new URL(txtfldAddressBar.getText())
```

If the string in the address bar is not a correctly formed URL, this should throw the appropriate error.

The second and third exceptions are triggered whenever the Algonquin web page is opened (for reasons yet to be explained). So if you want to check that you've caught this exception correctly, then call up the Algonquin home page in your browser. (Note however that not every browser seems to trigger this exception—results vary between students. You may need to deliberately `throw` these exceptions in your code to test that the `catch{}` block works correctly.)

Note that, since no harm is done by loading a page with errors on it—at worst something doesn't appear in the web page--the appropriate solution to such exceptions is simply to ignore them. That is, *do not* printout a stack trace, since a stack trace is exactly what you see when the exception is *not* handled by your code. If the exception generated by a web page has no real effect on the execution of your program, and if it doesn't affect the output, then the user doesn't need to see it: throw it away.

f. In Assignment 1, we were only able to *add* bookmarks to the Bookmarks menu. Now we wish to remove them as well. When the user *right clicks* on a bookmark, a context menu should appear asking the user if they wish to 'Remove Bookmark'. A click on the context menu then causes that particular bookmark to be removed. (Note that there should only be one context menu, and that it must be capable of removing any bookmark, regardless of the bookmark selected.) Recall also that all changes to the bookmark list must be saved to the bookmark.web file when the application is closed, so your code must update the underlying ArrayList whenever the bookmarks list is altered in any way. A very good source of information on using the context menu, and menus in general, is;

https://docs.oracle.com/javafx/2/ui_controls/menu_controls.htm

As an option, CustomMenuItem's getContent() method returns a Node object, which can be use to capture left and right mouse click events.

The information on context menus is at the very bottom of this page. As always, you can ignore the part of the sample code involving `addListener()`, since you'll already be capturing the left mouse click event with your existing `setOnAction()` handler code. But now you *will* need to modify your code to distinguish between a left mouse button click—which loads the selected URL into the web engine—and a right click that can remove the bookmark, to implement the event handler correctly.

g. Finally, your code needs to be fully documented using the Javadoc utility according to the Algonquin College Documentation Standard (ACDS). This was introduced back in Lab 1. See the 'Notes' section below for details.

III. Notes

a. As always, you must cite any and all web pages that you use in researching your code according to the guidelines outlined in Module 0 of this course, including those mentioned in this document. Failure to do so could result in a charge of plagiarism and disciplinary action being taken.

b. Follow the instructions in Lab 1 regarding the use of the JavaDoc generator. Additionally, review the document "Generating JavaDocs" which is available on Blackboard along with this Assignment.

A copy of the ACDS can be downloaded from Blackboard as well, along with the pdf *JavaDocRecommendations*.

Note that the ACDS mostly follows the Oracle documentation available at http://www.oracle.com/technetwork/articles/java/index-137868.html, "How to Write Doc Comments for the JavaDoc Tool." This web page provides a useful example of *what is expected of you* in documenting your code, which is reproduced in the text box above.

As noted in Lab 1, a common error is to forget to select the project before generating Javadocs. If you run the Java generator when only a single class is selected in the Eclipse Package Explorer, then only that class will have its comments appear in the JavaDoc doc folder—and you will lose marks despite the fact that you've documented your other class's methods correctly, but they just don't show up in the doc folder. Therefore, *after* you have generated your JavaDoc folder in Eclipse, but *before* you have zipped and shipped the *entire project* (as described below), open the doc folder and double click on index.html. This will generate a web page in Eclipse. Assuming you have selected the project folder *before* running JavaDocs, *all of your project's classes should appear in this web page in the pane on the left hand panel of the webpage*. If classes are missing, then go back and regenerate the JavaDoc doc folder again, this time being certain to click on the Project folder *first* since otherwise you will lose marks, since I can't mark what isn't submitted.

c. Get rid of any TODOs in your code; they're for you to do, not me: *I don't need to see them*. The only comments in your code should be the ones that explaining what each file, class, constructor, method and property does, according to the ACDS.

IV. Submission Guidelines

Your code should be uploaded to Blackboard (via the link posted) in a single zip file obtained by: (1) selecting the package name (`assignment2`) (2) right clicking on 'Export' (3) Make sure 'Archive File' is selected, and click Next (4) in the Archive File window make sure *all* of the program files are selected, including those provided to you, in the pane at right, and the 'Save in zip format' radio button is selected below (5) After creating the zip file, give it the following name EXACTLY AS WRITTEN

LastName_FirstName_Assignment2.zip

including the underscores, but with your last and first name inserted as indicated.
Note that

- As before, you do not need to include the bookmark and start-up page files with your submission. Your code should be robust enough that it creates and stores these files automatically if none is found in the default subdirectory.

- Since there are always some students who upload empty zip files, a good safety precaution is to take the .zip file you think you've just uploaded to Blackboard and open it on your laptop in a fresh location. Better still, start a new project in Eclipse, and load your zipped code into it. Make sure it works exactly as you expect after it gets transferred (you'd be surprised how often this simple test fails)

- You can upload as many attempts at Assignment 2 as you'd like, but only the final attempt is marked: all other attempts are ignored.

V. Bonus Marks (3 marks = 1% extra)

If you followed the Assignment 1 instructions correctly, you may have noticed that most of your Menu Item methods do pretty much the same thing. They (1)

instantiate a new menuItem with a given name (2) load an event handler, and (3) return the new menuItem. In this assignment, we add a new feature to this list: (4) add an accelerator key to the menu item.

This strongly suggests that we could tidy up a considerable amount of code by creating a single static method that took as parameters (1) a string that passes the name of the new menu item (like "Refresh" or "Exit") (2) a character to indicate the accelerator key, and (3) an EventHandler object, but using a lambda expression (to avoid having to instantiate space-consuming inner classes).

Similarly, each Menu method does three things, with little variation: it (1) instantiates a new Menu with a given name (like "File") (2) uses getItems().addAll() to load one or more menuItems, and (3) returns the new Menu from the method. Again, this suggests that a single static method, similar to the one for menu items, could be used to do the job (including adding the accelerator key).

Your task then is to build these two static methods, called `getMnuItm()` and `getMnu()`, and employ them in the Menus class to tidy up your repetitious code. You may find you need to:

(1) Make a few local properties global within the class, i.e. move them from inside the methods they currently reside in, and 'elevate' them up to global visibility in the class itself

(2) Create separate methods for the overridden `handle()` code. Where previously you referred to this code in lambda expressions, probably using something like `()->{…}`, now, to keep things tidy, you'll need to add this code as separate auxiliary methods inside the Menus class.

Nonetheless, you'll find that if you implement these two methods and employ them correctly, that you'll probably save upwards of 100 lines of code in the Menus class, with the benefit that the code will be both more compact and much clearer.

(And if you don't find your code works better as a result, then you're missing something. Contact me for further directions if you're stuck.)

Hint: you'll want to look into Java *varargs*, which will be useful in passing the new menuItems to `getMnu()`.

NOTE: If you attempt the bonus, clearly indicate that you have done so by adding a note with your submission to that effect. (Otherwise your hard work may get overlooked, and remain unmarked.)

Addenda

Revision 1.1

Some students have reported that the Algonquin web page generates an IllegalArgumentException, so this has been added to the list of possible exceptions that your code should check for. See the hybrid video(s) on Exceptions for details on adding multiple exceptions to the try/catch block.

Note that 'previous button should be disabled when the user is accessing the 0th web page in the history, while the 'next' button should be disabled when the user is accessing the last web page. (And, of course, the buttons must become enabled when all other pages are being accessed.)

Revision1.2

Rather than use a ContextMenu to handle right click events (on bookmarks) you might want to use instead the CustomMenuItem, with its getContent() method. This returns a Node, whose Events allow the programmer to distinguish left and right mouse clicks.

To learn more about exceptions, see the two hybrid videos (of ~35 minutes each). You may need the information on throwing specific exceptions to test your exception handlers if your browser does not 'see' the exceptions mentioned in the Assignment 2 document. (Apparently, some student's browsers aren't triggered by *any* of the exception mentioned in this document.)

For the MalformedURLException, see the URL class. Note that the URL(String) constructor throws MalformedURLException. Therefore, to test if the String in the address bar is a correctly formatted URL, try generating a new URL object with the URL(String) constructor first, i.e. `new URL(txtfieldAddressBar.getText())`. If the address isn't a valid URL, this should throw a MalformedURLException, which your code should catch.

Since it's frustrating for the user whenever a program ignores any kind of specific request, without providing feedback, your program should probably generate an Alert dialog to warn of a Malformed URL in this particular situation. (This is different from the Algonquin Web page situation, where the page loads correctly, and the problem is with the browser itself, not with the address the user is trying to enter.)