CS367 Programming Assignment 2
Lecture 4, Fall 2017
Due by 11:59 pm on Tuesday, October 24

In this page: Overview | Specifications | Handing in   Related pages: Assignments

Overview

Description

You have no doubt been in situations where you have seen the same sequence of images repeatedly displayed. For example, in a movie theater, before the previews start there is often a loop of images (advertisements, trivia questions, famous scenes, etc.) that is projected onto the movie screen. A screen-saver might cycle through a series of pictures from your last vacation. Information kiosks show sequence of pictures or instructions in a continuous loop. A series of drawings shown very quickly (and repeated over and over) results in the never-ending animation in the corner of a webpage you visit.

A Java program (in Eclipse archive form) that displays a sequence of labeled images may be found here. A major weakness of this program is that the content of the image sequence is "wired into" the program. That is, to change any detail of the sequence, the program must be rewritten.

In this assignment we will investigate a better approach. We will implement a simple GUI-based editor that allows a user to create, edit, save, and load a sequence of images (including the title of an image and length of time to display it). The control file the editor creates can then be uploaded and shown in a continuous loop.

It is typical for today's programmers to use ADTs that are already implemented as part of a class library or an API (e.g., Java's Application Program Interface). Understanding ADT *implementations* is still important to be able to make informed choices as to which ADT implementation is best suited for an application in terms of memory and time efficiencies. For example, knowing the tradeoffs between array and linked implementations of the `List` ADT helps us choose between Java's `ArrayList` and `LinkedList` classes. It is also important to be able to implement your own data structures when pre-built ones are not available or when standard ADTs (and their operations) are unsuitable for the intended application.

In this assignment we provide you with an interface for a specialized ADT called a **Loop**. In addition to the main editor application class, you will write a class that implements a Loop ADT using a circular doubly-linked chain of nodes, as well as iterator and exception classes needed to support your Loop ADT implementation.

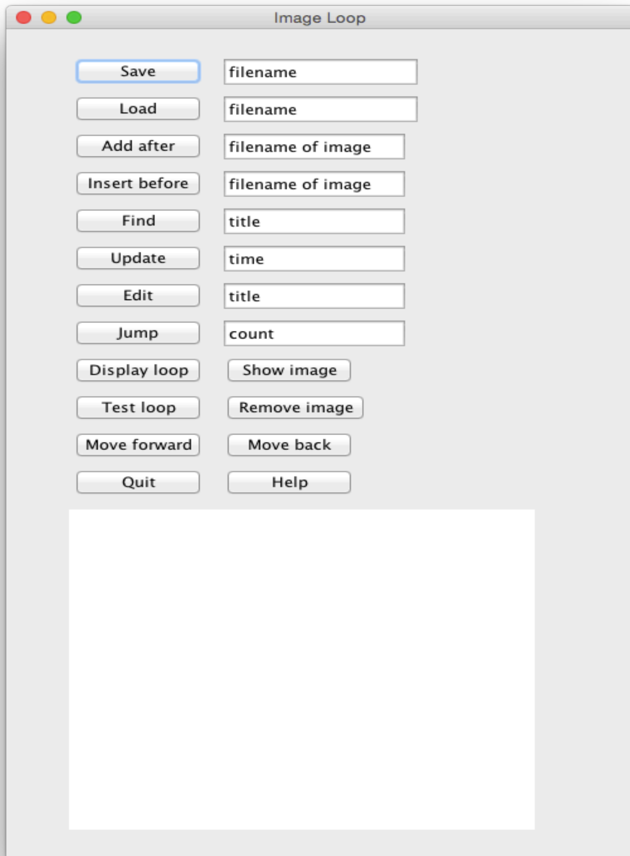
Goals

The goals of this assignment are to:

- Understand and implement a circular, doubly-linked chain of nodes.
- Gain experience working with Java references (i.e., pointers).
- Gain experience working with Graphical User Interfaces (GUIs).
- Gain experience working with advanced Java classes that implement windows and display image files

- Implement an iterator.
- Implement an exception.
- Gain experience writing classes that implement Java interfaces.
- Develop an application that processes input and editing commands.
- Get more practice with command-line arguments and I/O.

Program requirements



The `Image` Class

An `Image` object contains the name of the file containing the image, its title (possibly null) and a positive integer indicating the length of time (duration) the image should be displayed (in seconds). The `Image` class is provided for you (in the file `Image.java`). All the fields that store information about the image are private. The class has methods to retrieve and change information about a given instance of the `Image` class. It also has a method `displayImage` that creates a window and displays the contents of the image file. `displayImage` requires that all image files be placed in a top-level folder named `images`. The javadoc on the `Image` class provides the necessary information. You may **not** change the interface of `Image` class.

GUI-based Image Loop Editor

In the file `ImageLoopEditor.java`, you will create an empty LinkedLoop of `Images`. The GUI shown to the left will then be activated. Editing commands are initiated by pressing the appropriate button. Parameters are entered in a "text box" to the right of the button.

Here is a list of the editor commands you will need to implement. Some commands include a parameter; others do not. Thus the **Load** command needs a file name whereas the **Help** command does not. For simplicity all parameters are declared to be strings, even in cases where an integer is actually required.

Command	Parameter	Description
Help		Display information on available commands. (This command is already provided; you need not change it.)

Save	*filename*	If the image loop is empty, display "`no images to save`". Otherwise, save all the images to a file named *filename*, one image per line starting with the current image and proceeding forward through the loop. For each image, save the file name, the duration and the title. A null title (zero characters) is allowed; see example. If *filename* already exists, display "`warning: file already exists, will be overwritten`" before saving the images. If *filename* cannot be written to, display "`unable to save`".
Load	*filename*	If a file named *filename* does not exist or cannot be read from, display "`unable to load`". Otherwise, load the images from *filename* in the order they are given and set the current image to be the first image read from the file. You may assume that there is one image per line, that there are no blank lines, and that the file is not empty, i.e., it has at least one line. Each line contains a filename (a string), a duration (an integer) and a title (possible null) (see example). If a *filename* on a line is not in the `images` folder display "`Warning:` *filename* `is not in images folder`"
Display		If the image loop is empty, display "`no images`". Otherwise, display all of the images in the loop, starting with the current image, one image per line (going forward through the entire loop). Each line is of the form: `title [duration,filename]`.
Show		If the image loop is empty, display "`no images`". Otherwise, display the current image as a photograph, in a window with the image's title and for the specified duration.
Test		If the image loop is empty, display "`no images`". Otherwise, test the loop, starting with the current image, by displaying each image as a photograph in a window with the image's title and for the specified duration.
Move forward		If the image loop is empty, display "`no images`". Otherwise, go forward to the next image in the loop and display the current context (see note below).
Move back		If the image loop is empty, display "`no images`". Otherwise, go backwards to the previous image in the loop and display the current context (see note below).
Jump	*count*	If the image loop is empty, display "`no images`". Otherwise, jump *count* images in the loop (forward if $count > 0$, backwards if $count < 0$) and display the current context (see note below).
Remove		If the image loop is empty, display "`no images`". Otherwise, remove the current image. If the image loop becomes empty as a result of the removal, display "`no images`". Otherwise, make the image after the removed image the current image and display the current context

		(see note below).
Add after	*filename*	If the image loop is empty, add a new image with the given *filename*, a null title, and a duration of 5 seconds to the loop and make it the current image. Otherwise, add the new image immediately after the current image and make the new image the new current image. In either case, display the current context (see note below). If *filename* is not in the `images` folder display "`Warning:` *filename* `is not in images folder`"
Insert before	*filename*	If the image loop is empty, add a new image with the given *filename*, a null title, and a duration of 5 seconds to the loop and make it the current image. Otherwise, **i**nsert the new image immediately before the current image and make new image the new current image. In either case, display the current context (see note below). If *filename* is not in the `images` folder display "`Warning:` *filename* `is not in images folder`"
Find	*title*	If the image loop is empty, display "`no images`". Otherwise, find (by searching forward in the image loop) the first image whose title contains the given *string* (which may be quoted). If no image with a title containing *string* is found, display "`not found`"; otherwise, make that image the current image and display the current context (see note below). Comparison is case-sensitive, so "`rin tin tin`" does not match "`Rin Tin Tin`".
Update	*time*	If the image loop is empty, display "`no images`". Otherwise, update the duration for current image to the given *time* and display the current context (see note below).
Edit	*tilte*	If the image loop is empty, display "`no images`". Otherwise, edit the title for current image to the given *title* (which may be quoted) and display the current context (see note below).
Quit		Quit execution of the program.

Note on commands:
Displaying the current context means displaying the image (i.e., its title, filename and duration) immediately before the current image, the current image, and the image immediately after the current image (one per line). For example, if the current image is "`Lassie [dog2.jpg ,10]`", the image before current is "`Rin Tin Tin [dog1.jpg, 20]`", and the image after current is "`Bruiser [dog3.jpg, 15]`", your program should display:

```
        Rin Tin Tin [dog1.jpg, 20]
    --> Lassie [dog2.jpg ,10] <--
        Bruiser [dog3.jpg, 15]
```

The arrows "`-->    <--`" are displayed to highlight the current image. Since the GUI uses a variable width font, use a tab to align Image titles (the tab is set to be four characters wide).

However, if there are fewer than three images in the loop, do the following:

- If the loop contains only one image, such as "`Lassie [dog2.jpg ,10]`", then display:

```
    --> Lassie [dog2.jpg ,10] <--
```

- If the loop contains only two images, such as " Lassie [dog2.jpg ,10]" and " Bruiser [dog3.jpg, 15]", then display showing the current image first:

```
    -->  Lassie [dog2.jpg ,10] <--
         Bruiser [dog3.jpg, 15]
```

You will need to ensure the following:

- For commands that reference a filename, the string must start with at least one non-whitespace character and must contain only letters (a - z, A - Z), digits (0 - 9), underscores (_), periods (.), slashes (/) and dashes (-).
- For the **update** command, the time must be an integer.
- For the **jump** command, the count must be an optionally signed integer.

The Loop ADT

A Loop is essentially a circular List. However, unlike a List, a Loop does not have a beginning or an end and items within the Loop cannot be accessed using a position. Instead, the Loop has a current item and the ability to move forward or backwards. A Loop can be modified by removing the current item or by adding an item before the current item. The Loop ADT is represented in Java by the LoopADT<E> interface which is provided for you (see LoopADT.java). Complete information about Loop ADT operations can be found in the javadoc for the LoopADT<E> interface.

The LinkedLoop<E> Class

You will write an LinkedLoop<E> class (in a file named LinkedLoop.java) which implements the LoopADT<E> interface. In addition to the methods given in the LoopADT<E> interface, your LinkedLoop<E> class must have a constructor that takes no arguments and creates an empty loop. You may **not** add any other public constructors or methods.

The internal data structure used by your LinkedLoop<E> must be a circular, doubly-linked chain of nodes. You may **not** use Java's LinkedList class for this assignment. You must implement your own chain of doubly-linked nodes. To implement the circular, doubly-linked chain of nodes, you will need to use the DblListnode<E> class (provided for you; see DblListnode.java). Your LinkedLoop<E> class will have a field of type DblListnode<E> that references the node representing the current item. You may include other private fields that you find helpful, but no public fields.

Additional Classes

There are two other classes that you will need to write to go along with your LinkedLoop<E> class.

The EmptyLoopException class

The EmptyLoopException must be a **checked** exception. It is thrown by some methods (as described in the LoopADT<E> interface documentation). Your image loop editor class must handle them appropriately.

The LinkedLoopIterator<E> class

The LoopADT<E> interface includes an iterator() method that returns an Iterator<E> object. You must write a specialized iterator class, named LinkedLoopIterator<E>, which will

be used as the type of iterator returned by the LinkedLoop<E>'s iterator() method. The LinkedLoopIterator<E> class must implement the Iterator<E> interface (you do not need to implement the remove method - just throw an UnsupportedOperationException). You will need to write a constructor for your LinkedLoopIterator<E>; a package-access constructor that takes a DblListnode<E> as a parameter is suggested.

Testing

GUI testing interfaces are easy to master and use. But they can be frustrating if you aren't careful about recording the commands that you've entered. A bug can suddenly occur and you may not recall the exact sequence of commands needed to reproduce the bug.

A **text-based testing interface** represents each testing command as a single line of text. These commands can be grouped into a file. Input files can then be developed that test particular aspects of the program. These tests are easily repeated by running the tester again on a particular input file.

The table shown below associates a command name with each GUI command button. Case is insignificant. Moreover, a command can be abbreviated by any unique prefix. Thus quit can be abbreviated as q, but f can't be used because either find or forward might have been intended.

Help	help	**Save**	save	**Load**	load	**Display**	display
Show	show	**Test**	test	**Move forward**	forward	**Move back**	backward
Jump	jump	**Remove**	remove	**Add after**	add	**Insert before**	insert
Find	find	**Update**	update	**Edit**	edit	**Quit**	quit

Case may be significant in command parameters. Parameters may be quoted to make their content explicit.

Class TextImageLoopEditor is a subclass of ImageLoopEditor. It implements a text-based image loop editor. Commands are read, line by line, from the standiard input file. If a single command-line argument is provided, it is assumed to name a command line file. You may assume the file contains lines that start with at least one non-whitespace character and that the last line always is quit.

TextImageLoopEditor executes the same editing commands as the GUI editor (the PushXXX methods). You may use either editor in testing and debugging your program (sometimes it is easier to enter text than click GUI buttons). Be sure that command files work properly, since that's how we'll test your program.

How to proceed

After you have carefully read this assignment page and given thought to the problem we suggest the following steps:

1. The class collaboration policy allows you to do this program alone or with one partner. Make your choice!
2. Review these style and commenting standards that are used to evaluate your program's style.
3. You may use the Java programming environment of your choice, but we recommend Eclipse. You may want to review the Eclipse tutorial to learn the basics.
4. **Download** these provided files:
    - `DblListnode.java`
    - `Image.java`
    - `LoopADT.java`
    - `ImageLoopEditor.java`
    - `TextImageLoopEditor.java`
    - `images` (a folder containing 7 jpg photos)
    - `inlist.txt`
5. Write the `EmptyLoopException` class (one or two lines).
6. Incrementally implement and test the methods in the `LinkedLoop<E>` class (leaving the `iterator()` method until the end). You might want to use a driver program for your tests.
7. Implement and test the `LinkedLoopIterator<E>` class and the `iterator()` method of the `LinkedLoop<E>` class.
8. Complete the implementation of the `ImageLoopEditor` class, as described above, using a `LoopADT<Image>` object.
9. Test your editor using either the GUI-based editor (ImageLoopEditor) or the text-based editor (TextImageLoopEditor).
10. Develop test files for your editor as described in the testing section above. Try this sample input file, test.txt and make sure your program produces the same output as testOutput.txt. You can check if your program is producing the expected output using a file comparison utility. At the command line level, `diff` (or `diff3`) are often used. The command

    ```
    java TextImageLoopEditor test.txt | diff  - testOutput.txt
    ```

    will run the TextImageLoopEditor using `test.txt` as input. The output produced will be fed to `diff` and compared againt the expected output, `testOutput.txt`. The `diff` utility compares the two files character by character and displays any lines that differ. If anything gets displayed as a result of executing the `diff` command, then your program is not producing the expected output. No output means the the output is exactly as expected. ("No news is good news.")

    A variety of full-screen file comparsion utilities are also available (Mac, PC). You can capture the program output in the Eclipse Console window and enter it into a file. Then run your chosen file comparison utility.
11. For grading purposes, all class assignments will be run on a Unix/Linux box (using `javac` and `java`). You should verify that your program compiles and runs in this environment. If you are using a Mac this is easy -- use the `terminal` app located in the `utilities` folder. On Windows you can use the `cmd` app to obtain a command line and prompt. You can compile

your Java source files using `javac` in a terminal window as in this example:

```
javac *.java
```

and the run your program using `java` as in:

```
java TextImageLoopEditor test.txt
```

12. Submit your work for grading.

Handing in

What should be handed in?

Make sure your code follows the style and commenting standards used in CS 302 and CS 367.

Electronically submit the following files to the Program 2 Dropbox on Learn@UW:

- "`ImageLoopEditor.java`" containing your image display editor main program,
- "`LinkedLoop.java`" containing your circular, doubly-linked loop class,
- "`EmptyLoopException.java`" containing the checked exception used by the `LinkedLoop<E>` class,
- "`LinkedLoopIterator.java`" containing your `LinkedLoop<E>` iterator class, and
- "`*.java`" only if you implemented additional classes for your program

Please turn in only the file named above. There is no need to clutter your handin folder with files that are exactly the same as what we provided.

Last Updated: 8/11/2017   © 2008-2017 Charles Fischer and CS367 Instructors