

► Assignment 2

TODO Tracker (Part 2)

**Due: during your lab session not later than
Tuesday April 11th, 2017, at 11:59 p.m.**

Description:

This assignment builds on Assignment 1 by adding additional features to the TaskManager application, including

- an animated splash screen
- a file input dialog box
- allowing variable-sized .todo files to be used
- a simple menu to save and load files
- a warning whenever ToDo's have been changed, allowing the user to make decisions whether to save the new file or not before closing the program

Worth
7%
of your mark

Assignment 2

TODO Tracker (Part 2)

I. Create a new project.

- a. In Eclipse, create a new project called Assignment2
- b. Copy the package from Assignment 1 into the Assignment 2 src folder, and refactor as necessary. The simple step of backing up old files will save you much grief if you experience unexpected complications later during the development of Assignment 2; having a reliable version of your code to go back to is always good practice.

II. Make the following additions/modification to your Assignment 1 code

- a) **Add an animated splash screen:**
Currently, the `getDefaultScene()` provided in Assignment 1 only displays a single line of Text and does nothing else. We wish to liven this up with some animation. (Oracle has provided a nice summary of Java's animation features, located at <https://docs.oracle.com/javase/8/javafx/api/javafx/animation/package-summary.html>.)

JavaFX animations fall into two general categories: Timeline and Transition. (See <https://docs.oracle.com/javafx/2/animations/basics.htm> for another good overview.) In this course, we are only interested in Transitions, of which there are many to choose from, including

- ScaleTransition
- RotateTransition
- PathTransition

- FadeTransition
- SequentialTransition, and
- ParallelTransition.

For an introduction to Transitions, see any of the following:

<https://docs.oracle.com/javase/8/javafx/visual-effects-tutorial/basics.htm#BEIJJJB>

http://www.java2s.com/Tutorials/Java/JavaFX/1000_JavaFX_Transitions.htm

<https://rterp.wordpress.com/2015/09/01/creating-custom-animated-transitions-with-javafx/>

Note that transitions may be run either sequentially or in parallel (or in combinations of serial and parallel). See

<http://www.dummies.com/programming/java/how-to-combine-transitions-in-javafx/>

for details. Also: while most of these examples use rectangles for their animations and effects, any node will do, including the text object in which the word 'QuizMaster' is located.

Furthermore, JavaFX provides an Effects class, which allows you to produce results such as the following DropShadow Effect:



The list of Effects includes bends, blurs, glows, shadows, and reflections. For a complete list of Effects, see <https://docs.oracle.com/javafx/2/api/javafx/scene/effect/package-summary.html>

A good introduction to the DropShadow Effect, seen above, can be found at https://docs.oracle.com/javafx/2/visual_effects/drop_shadow.htm#CEGFADCA, from which the above graphic was taken. For information on the other Effects available

and their use, see the other web pages in this series.

Transitions and Effects may be combined with impressive results.

To complete the requirements for this section, you must, as a minimum, use any two of the following:

- ScaleTransition
- RotateTransition
- PathTransition
- FadeTransition

with either of the following

- SequentialTransition
- ParallelTransition

including any combination of the above. Additionally, you should provide at least one Effect with your Transition.

Your mission for this part of the assignment then is: apply animation and effects to the default 'TaskManager' scene, replacing the boring Text with a memorable splash screen.

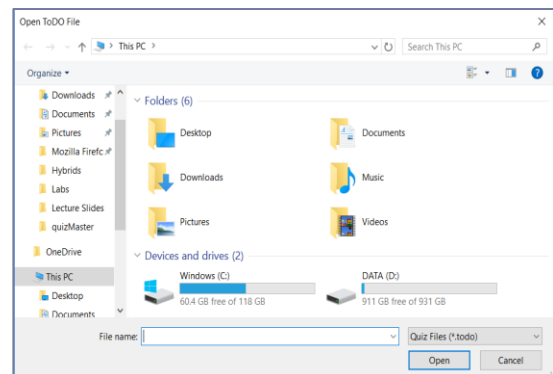
Put your code in a new method called `getSplashScene()` and use this in place of the current `getDefaultScene(...)`
Note that:

- As before, your new scene should use a mouse click to advance the program;
- You can loop the code so the animation repeats, if desired;

- Feel free to animate anything you wish, be it text, pictures, or shapes
- Don't get too caught up on this part; students sometimes waste hours playing with effects. If you love this stuff, take an animation course. For this Assignment, show you understand the basic concepts, and move on the rest of the program. Also, it is suggested that this be your last priority in this assignment, since the rest of the material, particularly the last sections, are more time-consuming.

b) Use FileChooser to load your default .todo file

Your existing code will need to be modified so that when the splash screen is clicked, rather than load the default .todo file as before, your code should execute the FileChooser dialog instead, using the code at the bottom of the page. The output is:



For additional information on using FileChooser, see the following websites:

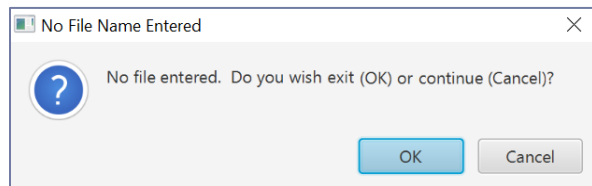
```
FileChooser fc = new FileChooser();
fc.setTitle("Open TODO File");
fc.getExtensionFilters().addAll(
    new ExtensionFilter("Quiz Files", "*.todo"),
    new ExtensionFilter("All Files", "*.*")
);
File todoFile = fc.showOpenDialog(primaryStage);
```

http://www.java2s.com/Tutorials/Java/JavaFX/0555_JavaFX_FileChooser.htm

http://docs.oracle.com/javafx/2/ui_controls/file-chooser.htm

Your program should continue execution *only* when the user has located and selected a valid .todo file using FileChooser. Therefore, if the user selects a non-valid .todo file, or selects the Cancel button, your code must generate a dialog box indicating that the choice made was not appropriate, and then prompt the user to continue—which takes him/her back to FileChooser—or exits the program, shutting down your JavaFX application.

- Use the `fileExists()` method provided in `FileUtils` to determine if the `todoFile` handle returned by `FileChooser` is valid or not.
- If it is not, call an `Alert` dialog, such as the one shown below.



- Rather than use the default location specified in Assignment 1 (whether it was in the `src` folder, the package, or the project) set the default path to your “D:” drive (adjust the existing `FileUtils` class if necessary). If your laptop lacks a D: drive, you can load your .todo file(s) into a USB and use it as D:. However you do this, make sure the `FileChooser` opens up in the D: drive, since that’s the folder the lab markers will have their .todo files in for testing purposes

See either of the following for further information on using `Alert` dialogs

<https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/Alert.html>

<http://www-acad.sheridanc.on.ca/~jollymor/prog24178/javafx7.html>

- If the user chooses to exit, the correct way to exit a JavaFX application is by calling `Platform.exit()`
- Write getters and setters in `FileUtils` to save the full (i.e. absolute) path from which the .todo was initially loaded. This will be put to use later in this assignment, when we wish to save a revised todo file back to its folder.
- Keep in mind that you will be opening `FileChooser` again from the `Open...` menu item, as indicated below. As always, practice good code reuse principles by writing your code so it can be called from a method, rather than rewriting the same code in a second location.

Your code should continue to loop correctly until the user either chooses a valid .todo file, or elects to exit the program.

When a valid .todo file is selected, then as before, your code loads the first element of the `ArrayList`, and execution proceeds.

c) **Adjust your code to allow variable-size arrays using `ArrayList`**

In this assignment, we’d like to be able to read in `ToDo` arrays of some size other than four. So we’ll first want to change our current primitive array of `ToDo`’s to a more-powerful `ArrayList()`. Therefore, you’ll need to adjust your code such that, wherever an array of `ToDo`’s currently exists, we use an `ArrayList` instead. This includes:

In FileUtils:

- `getToDoArray()` will need to return an `ArrayList` parameterized with `ToDo`'s, rather than the simple `ToDo[]` array that currently exists. And of course, your `ArrayList` will need to use the `add()` method, rather than increment through a `for` loop with a counter;
- Of course, you can now remove the `NUMBER_OF_TODOs` constant, since you can always determine the size of an `ArrayList` from its `size()`.

In TaskManager:

- The declaration of the private `toDoArray` property will need to change to an `ArrayList`;
- The setters and getters for the `toDoArray` will need to change accordingly;
- `getToDoScene()` will need to be adjusted to receive an `ArrayList`.

Of course, depending on how you implemented your code, there may be other areas requiring your attention. But generally speaking, once you have transitioned your Assignment 1 code from arrays to `ArrayList`, there should not be any drastic changes. After all, the program deals mostly with `ToDo` objects, and these are the same whether stored in an array or an `ArrayList`.

d) Add a menubar to the application

By convention, the first item in any menubar is `File`. So your top (and only) menu item will be `File`. The five menu items under `File` must be:

```
Open...    // to call the FileChooser()
           // and load a new .todo File
```

```
Save...    // to save a modified TODO
           // arraylist back to its file
Add ToDo...
           // adds a new ToDo to the
           // current arraylist
Remove ToDo...
           // delete the current ToDo
           // element
Exit       // to quit the program.
```

Regarding these five menu items, note that:

- `Open...` calls `FileChooser` as before. If a new valid `.todo` file is selected with `FileChooser`, then it should be loaded into `TaskManager` as in step (b) above. However, if any element of the current `ToDo ArrayList` has changed (as indicated by any attempt to edit the contents of the center pane – see part(e) for details on implementing this feature), then the user needs to be prompted as to whether they wish to save the current `todo` file before loading the new `todo` file.
- `Save...` saves the current `ToDo ArrayList` item back to its original file. (We'll assume the user does *not* wish to change folders or the file name. Hence there's no need to use a `FileChooser` `showSaveDialog`, as outlined in the `FileChooser` example website in part (b) above. Be sure to use the absolute path name—returned by the getter you constructed above—to save the file.) As with `Open...`, if any changes have been made to any `ToDo` element, the user should be prompted whether they wish to save the changes made back to the source `.todo` file.
- `Add ToDo...` allows the user to insert a new (empty) `ToDo` after the current `ToDo`, and displays the various fields for user input in the

center pane. So your code will need to do two things: insert a new `ToDo` into the `ArrayList`, and load a center pane with empty boxes.

To load empty boxes into the center pane, use the default constructor to instantiate a new `ToDo` object with nothing inside: no Title, no Text, no priority. (You can use the same `ToDo` insert a new `ToDo` into the `ArrayList`). Only the `remove` property should be set `true` for reasons which will become clearer in the next section (Of course, the date stamp will be set as well.) Then load this blank `ToDo` into the center pane using `getToDoScene()` ;.

- iv. `Remove ToDo...` deletes the current `ToDo` element from the `ArrayList`. Your code should first prompt the user with a dialog window asking the user to confirm the deletion
- v. `Exit` causes the program to terminate. As before, if any `ToDo` element has changed, the program prompts the user to see if they wish to save a fresh copy of the `ToDo ArrayList` back to the file first.

Note that for each of these items, choosing the Cancel button (which returns `null`) simply causes the dialog box to disappear, leaving the center pane unchanged at the current `ToDo`.

See the section that follows for advice on implementing the code associated with these features.

Several web sites explain how to add menu items to an application. The following may prove useful:

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

http://www.java2s.com/Tutorials/Java/JavaFX/0560_JavaFX_Menu.htm

<http://www.javaworld.com/article/2074463/core-java/-pure-java--javafx-2-0-menus.html>

Note that:

- The menubar is attached to the top region of the `Borderpane`, which was left empty in Assignment 1. Thus it should not be affected by loading a new `ToDo` object in the center pane (that is, assuming you correctly followed the instructions in Assignment 1)

- e) **Test to see if a `ToDo` has been changed and issue a warning if it has not been saved first, prior to exiting the program or loading a new `.todo` file**

As a final feature your code, it needs to be able to prompt the user if the application is shut down, and any changes have been made to the underlying `ToDo`'s have not been saved back to the original `.todo` file.

This includes the possibility that the user attempts to shut down the window (i.e. the `primaryStage`) without saving any `ToDo` modifications to the file first.

To trap the windows close event, you'll need to write an event handler and load it using

```
stage.setOnCloseRequest(
    WindowEvent we){ }
```

For examples, see the following

<http://www.java2s.com/Code/Java/JavaFX/StageCloseEvent.htm>

<http://stackoverflow.com/questions/22576261/how-get-close-event-of-stage-in-javafx>, but note that only the final respondent got it right. (The guy who writes: "I got the answer...", didn't)

Note that this 'set()' method takes a `WindowEvent` rather than the usual

ActionEvent you're familiar with. But the principle is exactly the same.

To implement the actual code, you'll need to do make the following changes to TaskManager:

- i. We will be interested in the contents of the two Text boxes, the Textfield, and the radio buttons, corresponding to the Title, Due Date, Subject and priority. To simplify matters below, you'll probably want to add private properties corresponding to each of these items in TaskManager, and then add getters and setters so that you can readily access the contents of these nodes without having to 'operate' on the contents of the center pane. (There are better ways to do this, but adding these properties to our code is easier than rewriting your existing code to make these nodes more accessible.) Therefore, use getters and setters to access these Nodes from this point onward.
- ii. We will use the ToDo object's `remove` property to determine whether the current ToDo object is 'dirty' or not. ('Dirty' is the term used whenever a file's contents have been changed. Thus whenever you edit a Word document—even if you immediately reverse the changes—the 'dirty bit' is set, and this tells the application to prompt to see if the user wishes to save the changes.) So your code should execute the `setRemove(true)` method of the current ToDo object to signal that the contents of the center pane have been changed. *If the `remove` bit is set, then the assumption is that the contents of the current ToDo will need to be updated, based on changes to the nodes in the center pane.* (Yes, `dirty` would have been a better name for what this bit of information does; but `remove` works just as well.)

The criterion by which we judge if a particular ToDo has been made 'dirty' is that some ActionEvent has occurred on the Title text box, the Subject textfield box, the Date text box, or the radio buttons. So any attempt to change any of these three nodes—even a simple mouse click—will be taken as an indication of the intention to make changes to the underlying ToDo element whose contents are loaded into the center pane.

So for each of the graphics objects just listed, your code should add an event handler, and inside call the current ToDo's `setRemove(true)` whenever an Action Event is triggered on any of these nodes.

- iii. Whenever the user clicks the First, Back, Next, or Last buttons, selects a menu item, or attempts to close the window—anything that moves the program away from the current ToDo element—your code needs check the current ToDo object to see if `isRemoveSet()` is now `true`. If it is `false`, then we assume the current ToDo has not been edited (i.e. the nodes in the center pane have not been changed), and we can safely go about carrying out the requested operation. If the `remove` bit is `true`, then we need to
 - a. Prompt the user to see if they wish to save the changes to the ToDo element currently being displayed in the center pane (use a dialog box for this),
 - b. If they do not, then ignore the changes to the center pane and follow the instruction selected (i.e. do Next, Last, menuitem, etc.) But if the user chooses to save the changes then call the following method.
- iv. `saveCenterPaneContents2ToDo()` uses the getters just added—for the

Textboxes, Textfield and radiobuttons—takes their contents and loads them into a temporary `ToDo` object using the `ToDo()` constructor. Use this `ToDo` to replace the existing `ToDo` in the `ArrayList`, which will now contain the revised information according to the new contents of the three center pane text objects.

When you call the `ToDo()` constructor, note that the `empty` and `completed` bits are all set to `false`. (The `remove` bit is still `true`, since the `ToDo ArrayList` itself is now ‘dirty’.)

When you’re finished executing this method, be sure to complete the selected method that originally called `saveCenterPaneContents2ToDo`.

- v. Create the function `isToDoArrayListDirty()`, which takes the current `ToDo ArrayList` (use the getter you created in Assignment 1, which was adapted for `ArrayLists` in part (c)) and loop through each to see if its `remove` bit is set (using the `isRemoveSet()` getter). As soon as `isRemoveSet()` returns `true` for any of the `ToDo`’s, this function should break and return `true`.

`isToDoArrayListDirty()` is used when your program is (a) about to exit, or (b) about to open a new `.todo` File using the `Open...` menu item, and needs to decide whether or not to back up to revised `ToDo ArrayList` to the original file. If the `ToDo ArrayList` is dirty, then you’ll need the next function before exiting the current `.todo` file...

- vi. `setToDoArrayListToFile()` takes a `ToDo ArrayList` and a file name—the current `ToDo ArrayList` and the absolute path belonging to the current opened file—and saves the new `ArrayList` to that file using File Output

Stream and Object Output Stream. As with `getToDoArray()` method, this method should be a static method of the `FileUtils` class. Since each `ToDo` object in the current `ToDo ArrayList` is assumed to be up to date at this point, there’s no need to prompt the user to check if they wish to save the new information to the file: the assumption (for our purposes) is that having selected ‘Yes’ whenever a `ToDo` was updated, and so the actual file update will happen automatically.

Note also: since this method is the last step in the chain, the one that saves any changes made by the user, it should also perform some housekeeping on the `ToDo ArrayList` itself. In particular:

- (a) remove any empty elements (check the `isEmpty()` method for each `ToDo`); save only non-empty `ToDo`s.
- (b) any `ToDo` element still listed as ‘dirty’ should be ‘sanitized’. Thus apply

```
if (thisToDo.isRemoveSet())
    thisToDo.setRemove(false)
```

prior to saving the `ToDo ArrayList` back to its `.todo` file.

III. Notes

As before, you have considerable latitude in choosing the details of your implementation. However, the methods listed above are considered to be essential to your submission for this assignment. (This being said, some students may have implemented their code in ways which allow for departures from the prescribed methods above. If you’re one of those students, please feel free to ask any questions about any variations that you’d like to explore.)

In Assignment 1, the emphasis was on getting the

program to work according to the requirements provide, with less emphasis on writing elegant code. In this assignment, more marks are accorded for programming style. In particular, your code should be modular, breaking long complicated stretches of code into concise methods that perform specific functions. Use the code provided in previous labs, lectures and assignments—as well as good common sense and a feeling for aesthetics—as your guide. And always practice code reuse.

For example, the `Exit` menu item will perform the same effect as shutting down the application using the `setOnCloseRequest()`. So both should call the same method. Similarly, the `Open...` menu item will need to use the code employed in the `Save...` menu item if the user has updated the current `ToDo ArrayList` and wishes to save it. So write your code with both modularity and connectivity in mind.

IV. Submission Guidelines

Your code should be shipped in a single zip file obtained by: (1) selecting the *entire project* in the Eclipse Package Explorer window (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 your files are selected in the pane at right, and the 'Save in zip format' radio button is selected below, and (5) Give your zip file the following name EXACTLY AS WRITTEN

LastName_FirstName_Assignment2.zip

including the underscores (where *LastName* and *FirstName* are your last name and first name.)

Addenda:

Check Blackboard periodically for clarification of problems as they arise.

Assignment 2 Marking Guide	
Requirement	Mark
<p><i>Code structure:</i></p> <p>The program is correctly modularized. Rather than having all the code loaded into a single method, the 'divide-and-conquer' strategy has been employed to make the code more readable. This includes 'off-loading' longer chunks of code to static methods and instantiated classes, where appropriate</p>	/5
<p><i>Followed the Instructions:</i></p> <p>Followed the instructions in Section II of this document (including any late additions/changes), and in particular used the methods correctly, as described below:</p> <p>Section II(a): Animation. Marked out of 3. You must supply transitions and effects as indicated</p> <p>Section II(b): FileChooser. Marked out of 4. Your code successful loads the file selected, and exits the loop when the user wishes to cancel opening of the application</p> <p>Section II(c): ArrayList. Marked out of 3. Use an ArrayList in place of the arrays that currently exist in the code, and use them correctly throughout the program</p> <p>Section II(d): MenuBar. Marked out of 6. All menu items work as indicated, along with the Windows exit</p> <p>Section II(e): ArrayList Backup: Marked out of 8. Any revisions to the ToDo ArrayList, as indicated by attempts to edit the center pane material (excluding radio buttons) are successfully backed up to the .todo file as required.</p>	/24
<p><i>Compiles and executes correctly:</i></p> <p>The program compiles, i.e. there are no 'red dots' in the margin of any class loaded into Eclipse. Code appears to execute correctly, i.e. no obvious defects. Note that a program which does not execute is difficult to mark accurately. In addition to marks being lost in this section, a considerable number of marks will be lost from the above section, which, while the code may be correctly written, is not provably so if the code does not execute.</p>	/3
<p><i>MINUS:</i> late penalty, failure to cite authors, etc.</p>	
<p>Total:</p>	/32