W. P. CAREY
SCHOOL *of* BUSINESS
ARIZONA STATE UNIVERSITY

CIS 494 – Business Systems Development with Java – Spring 2015

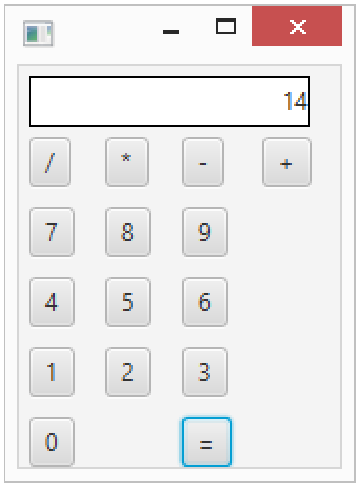
In-class Exercise 6: Event Handling

Due on Blackboard: Tonight, February 25 by 11:59 PM

The purpose of this exercise is to gain basic proficiency in creating implementing event handling within Java 8.

Implement a basic four function calculator with event handlers for all buttons representing numbers and operators.

Sample Output



Declare all GUI elements as ***instance variables***, i.e. within the class block, so the variables are in scope in all methods.

Use either Eclipse's predictive guessing or Javadocs to look up exact method names! Get in the habit of locating appropriate methods! :-)

- Declare the following controls.

  - 10 number Buttons going from button0 to button9
  - 4 operator Buttons, buttonPlus, buttonMinus, buttonDivide, and buttonMultiply
  - 1 Button, buttonCompute.(=)
  - 1 Label, answerDisplay
  - 1 BorderPane, borderPane
  - 1 GridPane, buttonPane

- Declare variables to store number1, number2 as integers and operatorText (it will store whether user selected +, -, etc.) as strings. *You can choose any other more sophisticated methodology or data structure if you like!*

Method: buildScene
Return type: Scene

- Instantiate borderPane using the BorderPane constructor and set give it a padding.
- Instantiate buttonPane using the GridPane constructor
- Instantiate all 10 buttons for the numbers. Supply the Button constructor with the string to display on the face of the button.
- Instantiate all four operator buttons and the compute button and supply the constructor with the string to display on the face of the button.
- Make a call to the setActionHandlers() method. *You will write this method next.*

- Instantiate the answerDiplay Label by calling its constructor.
- Set the Alignment property of the Label using the setter. The Pos class has static constant variables you can use to provide arguments. You preferably want text right-aligned, so you can provide it with Pos.CENTER_RIGHT.
- Set the minimum height and width of answerDisplay using setter methods. *Experiment with different values at the end based on your screen resolution, etc.*
- Set the style of the label according to how you want it to look (background color, border color, etc.)
- Set the Label as the Top element of the BorderPane. *You can alternatively create a new StackPane (or any other type of Pane you like) and add the label to that pane. Then you would add that pane to the Top of the BorderPane. Experiment with looks at the end!*

- In succession, add all the buttons to your buttonPane, which is a gridPane. You will need to provide it with node name (your buttons are nodes), column number, and row number. *You will need to work out column and row numbers!*
- Set the buttonPane as the Center element of your BorderPane.
- Create a new Scene and add the BorderPane to the Scene
- Return the scene object!

Methods for Number Button Event Handlers

- Create 10 individual methods that will each serve as the event handlers for each of the number buttons. You should name these methods, process1(), process2(), and so on.
- Each method needs only one line of code in it.
    - Each method should take the text of the answerDisplay Label and append to it using concatenation the specific number that the event hander handles. Then put that new string back into the Label to be displayed.

      For example, for the event handler for number 1, you obtain whatever text is in the label by using the getText() *getter* method. Append "1" to it using string concatenation and put the string back in by using the setText() *setter* method. So, if the display showed "52" and the user clicked the "1" button, the display would then show "521" after being updated.

Methods for Operator Button Event Handlers

- Create 4 individual methods that will each serve as the event handlers for each of the operator buttons. You should name these methods, processPlus(), processMinus(), and so on.
    - Set the oepratorText to show what operator the user clicked on. For example, if the user clicked on "+", operatorText should have "+" in it.
    - Parse the text of answerDisplay and store it in number1.
    - Reset the text of answerDisplay to show a blank string.

Method: compute()
Return type: void

- Parse the text of answerDisplay and store it in number2.
- Create a total variable.
- Use a switch statement and switch on operatorText. Create four cases for each of the operators. Within each of the cases, perform the calculation on number1 and number2 using the specified operator and store the result in total.

  For example, if operatorText stores "*", you should multiply number1 by number2 and store the product in total.

Method: setActionHandlers()
Return type: void

This method will “set” or “register” the event handlers. We already have a method that we want to be invoked when the clicks on a button. We need to essentially let the system know which method needs to be called.

This can be done by creating a new class that implements the Event Handler <ActionEvent> interface. These could be outer classes, inner classes, or anonymous class implementations. Either way, this would be a lot!

Instead, for this assignment, use *Lambda* expressions, which will simplify action listener specification.

- For each Button (15 in total!), invoke the setOnAction method. Provide it with the argument in the following structure:
  e -> eventHandlerMethodName()

That’s all! This one line of code will replace all the ActionEvent interface implementation!
Aren’t Lambda expressions nice!?

Method: start()

In your start method, call your buildScene() method which returns a scene object. Pass the return value to the setScene method of the Stage object.

Finally, show the scene!

Optional Enhancements

1. Create a Backspace button! Use the String class and select the substring that you want to display to emulate deleting a character.

2. Improve the look of the application. See if you can create a large-sized Compute button and align it properly. You might need to add another nested pane to get your button to display correctly without it displacing other buttons (which would happen in the GridPane).

Optional Advanced Implementations:

1. Implement the Compute button using the traditional handle method using ActionEvent parameters. Use an ***inner*** class, so that the inner class has access to the instance variables of the outer class.

2. This implementation of the Calculator methods is highly inefficient. It is full of repetitive code. A couple of improvements can be made:

> 2.1 Create ONE unified event handler for <u>all</u> number buttons and ONE unified event handler for <u>all</u> operator buttons.
>
> The problem: how will you know what button was clicked since you need to do slightly different things for each button?
>
> The “e” in “e -> eventHandler()” is an Event parameter (of type ActionEvent). You can pass it to the method as an argument (your event handler method will need accept an ActionEvent as a parameter).You can then call e’s .getSource() method, which will provide you with the source of the event, i.e. the button itself. You can *downcast* the return value as a Button and get access to the text of the button! Use a switch statement on the button text. Thus, you can create a unified event handler for all buttons.
>
> 2.2 Use arrays of Button objects
>
> The number of button objects all declared independently is all inefficient. You can transform all buttons into a single array of buttons. Once you transform all buttons (number buttons only), you can gain efficiencies by adding them to the panes using a *for* loop. You can also implement logic to come up with the row and column numbers!

<u>Submitting Files</u>

Submission should be made using a zip file that contains the entire Eclipse project folder. You will need to ***zip the entire project folder***. The folder will automatically contain the class source files as well as the compiled .class files.

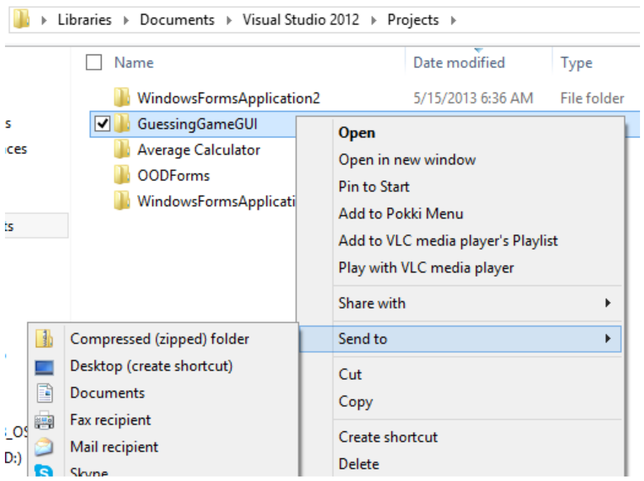
ZIP file should be named: **AX.zip or IC.zip**

where X is the in-class assignment number, e.g., A1.zip is the submission file for Assignment 1 and IC1.zip is the submission file for InClass1.

Note: *The ZIP filename is <u>independent</u> of the Project name. Do not name your project A1.*

How to Properly ZIP and Submit Your Project Files:

- Go to the folder within {Eclipse Workspace}\
- ZIP the entire top-level folder for your project by right-clicking your project folder and selecting Send to | Compressed (zipped) folder.
- Finally, submit the ZIP file using the submission link on Blackboard by the due-date and time listed on the assignment. Upload the ZIP file.



Using built-in windows zip tools: http://windows.microsoft.com/en-us/windows/compress-uncompress-files-zip-files

Verify your files BEFORE and AFTER submission:

- Check for actual class files being present in the folder before you zip it.
- ***Ensure that you are not zipping a short-cut to the folder.***
- After zipping, check file size. A file size under 4K likely does not contain all the files.
- Unzip, extract all files, and verify you see actual files, not a solitary short-cut.
- Uncompress your zip file before submitting and verify that files are present.
- Make sure you have ***submitted*** your file and not just saved a draft on Blackboard. ***A blue clock indicates a submission in progress, i.e. a draft, not a submission. The draft is accessible only to you. You will get a ZERO if you only ever save a draft on Blackboard and never submit your files.***

- Download your zip file after submitting, uncompress, and again verify that your files are present. Test your files in Visual Studio after uncompressing.

This takes an extra couple of minutes. Please do it if your grade is important to you. If you do this, you will not end up submitting a bad file. If you submit an empty file, or one containing only a shortcut, or a bad zip file, or a bad project file, you will receive a score of zero and your only recourse will be to do the makeup assignment at the end of the semester.