Application Program Development

Segment : Standard Dialog Boxes

Mahboob Ali

Outcomes

- Understanding Menus in JavaFX
- Different options to work with Menus.

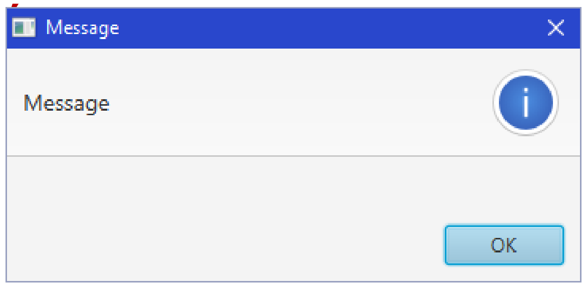
Standard Dialog Boxes

- If a main application window has too many components on it, it will look cluttered and it will not be simple and easy to use.
- It is a good idea not to display components on your window if they are not needed at that time. For example, a main application may not want to display name, address and phone number fields until the user has selected some action that requires that information to be entered.
- Usually, this information is placed in a different window that "pops up" when needed.

  *A **Dialog Box** is a secondary window (i.e., not the main application window) that is used to interact with the user ... usually to display or obtain additional information.*

- So ... a dialog box is another window that can be brought up at any time in your application to interact with the user.
- In JAVA FX, there are some "standard" dialog boxes that are pre-made.
- The programmer just needs to specify a few settings and what he/she wants to appear in the window. Then JAVA FX does the rest.
- The **Alert** class is used to represent a standard dialog box in JAVA FX.

```
Alert alert = new Alert(Alert.AlertType.INFORMATION);
```

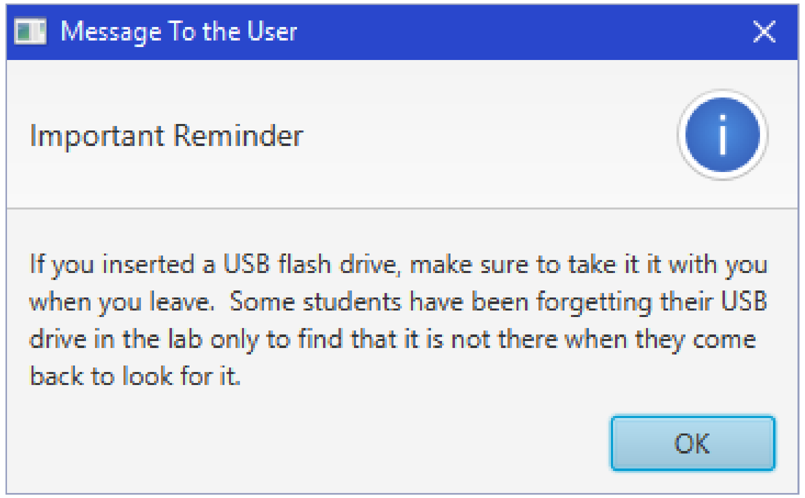


- The dialog box is then shown by using:

```
alert.showAndWait();
```

- It is a simple, plain dialog box that remains open until the user presses the **OK** button.
- However, the dialog box is customizable.
- We can alter the title of the window, the "header" portion of the dialog box (i.e., the text to the left of the icon) as well as add some additional context-related text just above the OK button.

```
alert.setTitle("Message To the User");
alert.setHeaderText("Important Reminder");
alert.setContentText("If you inserted a USB flash drive, make sure " +
                     "to take it with you when you leave. Some " +
                     "students have been forgetting their USB drive" +
                     " in the lab only to find that it is not there" +
                     " when they come back to look for it.");
```

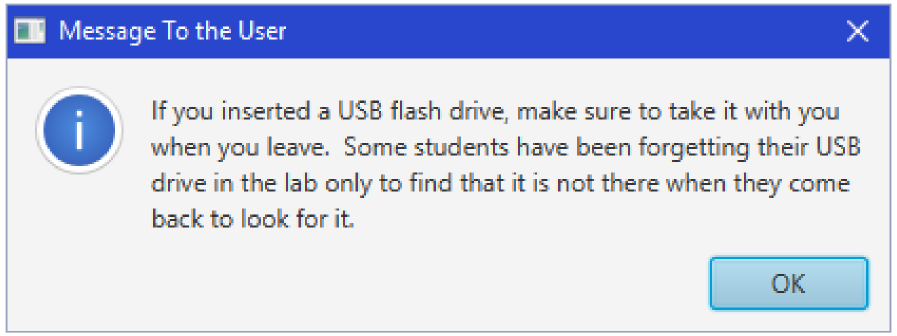

Message To the User

Important Reminder

If you inserted a USB flash drive, make sure to take it it with you when you leave. Some students have been forgetting their USB drive in the lab only to find that it is not there when they come back to look for it.

- You can also eliminate the Header text by setting it to null:

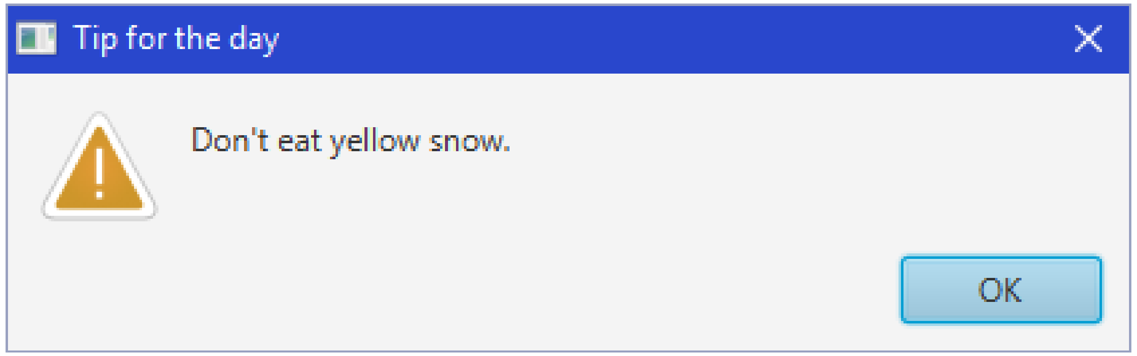
```
alert.setHeaderText(null);
```


Message To the User

If you inserted a USB flash drive, make sure to take it with you when you leave. Some students have been forgetting their USB drive in the lab only to find that it is not there when they come back to look for it.

- You can change the icon from being an "information" icon to that of being a "warning" icon or an "error" icon simply by altering the static value used when creating the Alert:

```
Alert alert = new Alert(Alert.AlertType.WARNING);
alert.setTitle("Tip for the day");
alert.setHeaderText(null);
alert.setContentText("Don't eat yellow snow.");
```


Tip for the day

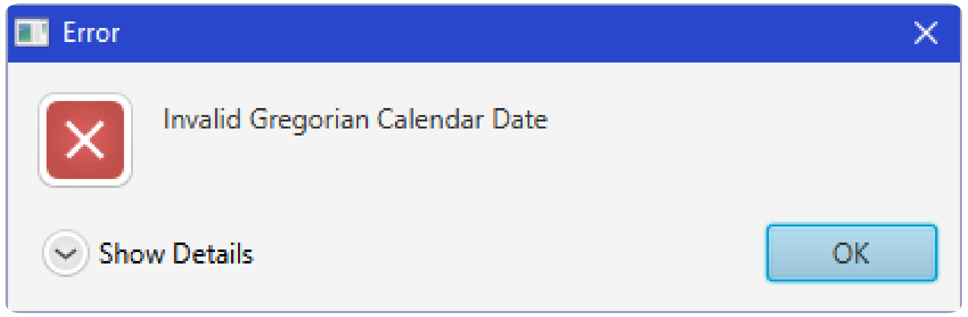
Don't eat yellow snow.

- In addition to these simple messages, we can also have a message dialog come up with a lot of text within it by adding a **TextArea** to the Alert.

```
Alert alert = new Alert(Alert.AlertType.ERROR);
alert.setTitle("Error"); alert.setHeaderText(null);
alert.setContentText("Invalid Gregorian Calendar Date");
Label label = new Label("Date entered:");
TextArea textArea = new TextArea(new GregorianCalendar().toString());
textArea.setEditable(false); textArea.setWrapText(true);
textArea.setMaxWidth(Double.MAX_VALUE);
textArea.setMaxHeight(Double.MAX_VALUE);
GridPane.setVgrow(textArea, Priority.ALWAYS);
GridPane.setHgrow(textArea, Priority.ALWAYS);
GridPane expandableContent = new GridPane();
expandableContent.setMaxWidth(Double.MAX_VALUE);
expandableContent.add(label, 0, 0);
expandableContent.add(textArea, 0, 1);
alert.getDialogPane().setExpandableContent(expandableContent);
```

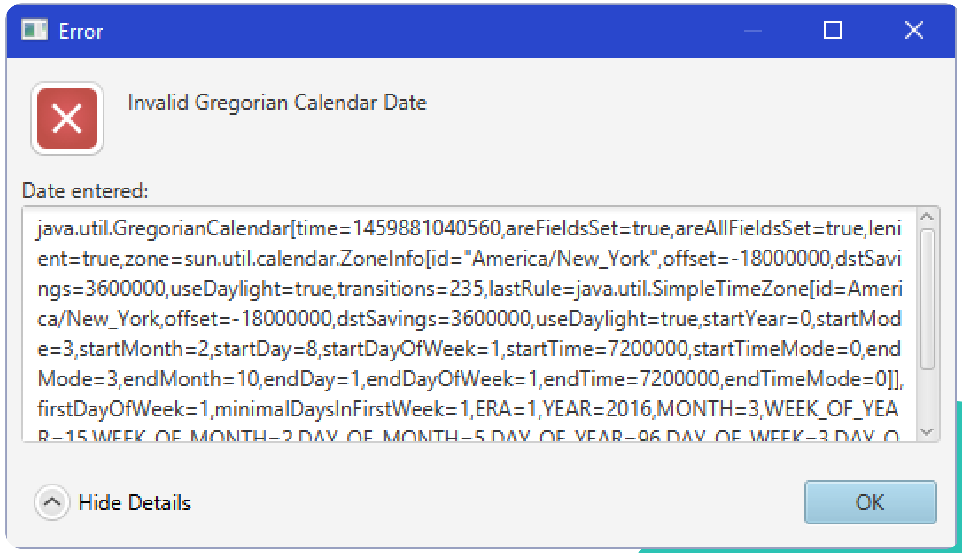
- Notice that we are just using a standard error dialog but that we are adding a **GridPane** with a **Label** and a **TextArea**. We added the data from today's date, created from a new **GregorianCalendar** object.

- The **setExpandableContent()** method allows us to add that extra **GridPane** to the dialog box in a way that will allow the user to Hide or Show it as extra detail. Here is what the dialog box looks like with the data hidden.



Invalid Gregorian Calendar Date

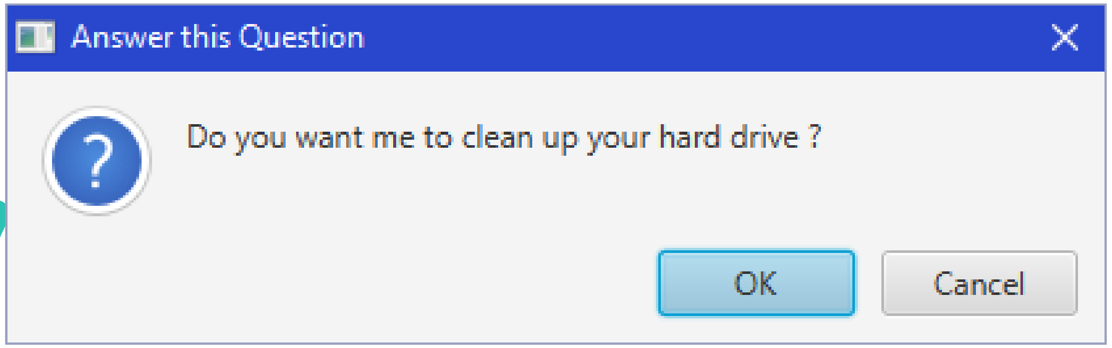
- And here is what it looks like with the expanded data being shown



Invalid Gregorian Calendar Date

Date entered:

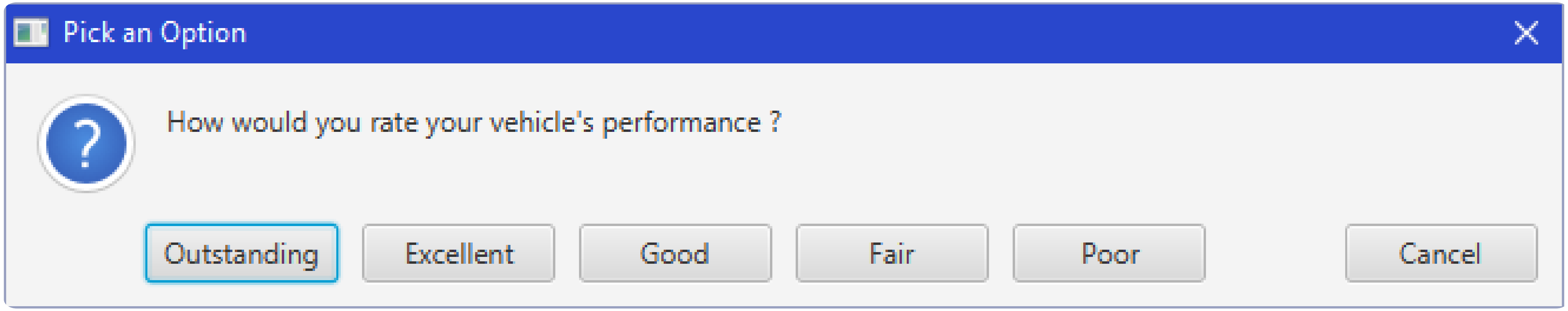
```
java.util.GregorianCalendar[time=1459881040560,areFieldsSet=true,areAllFieldsSet=true,leni
ent=true,zone=sun.util.calendar.ZoneInfo[id="America/New_York",offset=-18000000,dstSavi
ngs=3600000,useDaylight=true,transitions=235,lastRule=java.util.SimpleTimeZone[id=Ameri
ca/New_York,offset=-18000000,dstSavings=3600000,useDaylight=true,startYear=0,startMod
e=3,startMonth=2,startDay=8,startDayOfWeek=1,startTime=7200000,startTimeMode=0,end
Mode=3,endMonth=10,endDay=1,endDayOfWeek=1,endTime=7200000,endTimeMode=0]],
firstDayOfWeek=1,minimalDaysInFirstWeek=1,ERA=1,YEAR=2016,MONTH=3,WEEK_OF_YEA
R=15,WEEK_OF_MONTH=2,DAY_OF_MONTH=5,DAY_OF_YEAR=96,DAY_OF_WEEK=3,DAY_O
```


Answer this Question

Do you want me to clean up your hard drive ?

- In addition to showing just simple messages, we can have dialog boxes that allow the user to make a simple decision.

```
Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
alert.setTitle("Answer this Question");

alert.setHeaderText(null);

alert.setContentText("Do you want me to clean up your hard drive ?");

Optional<ButtonType> result = alert.showAndWait();

if (result.get() == ButtonType.OK){

        System.out.println("OK, I'm erasing it now ..."); }

else { System.out.println("Fine then, you clean it up!"); }
```

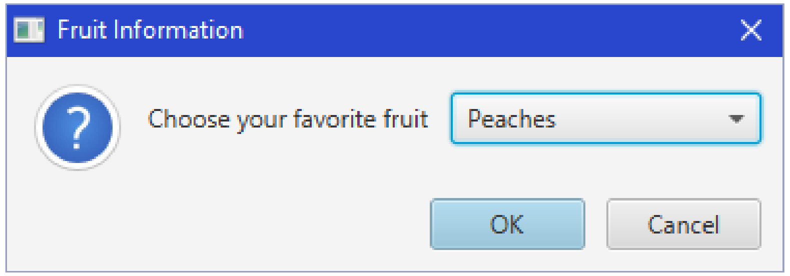


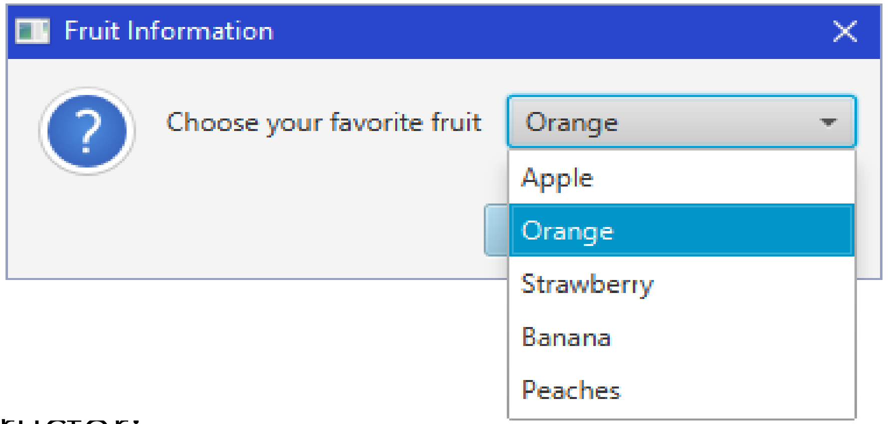
- Notice that we can ask for the result from the **showAndWait()** method.
- We can use the **get()** method to find out which **ButtonType** was pressed (i.e., OK or CANCEL), and then act accordingly.
- If the user closes the window by pressing the X on the top right, this is considered as CANCEL in our code.
- We can even customize the buttons on this confirmation dialog box.
- Consider a bunch of options like this the picture above.

- We can do this with a standard confirmation dialog box where we replace the buttons with our own choices:

```
Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
alert.setTitle("Pick an Option"); alert.setHeaderText(null);
alert.setContentText("How would you rate your vehicle's performance ?");
ButtonType[] buttons = new ButtonType[6];
String[] buttonNames = {"Outstanding", "Excellent", "Good", "Fair", "Poor"};
alert.getButtonTypes().setAll(); // Erases the default buttons
for (int i=0; i<5; i++) {
       buttons[i] = new ButtonType(buttonNames[i]);
       alert.getButtonTypes().add(buttons[i]); // Adds this new button
}
// Add a cancel button
alert.getButtonTypes().add(new ButtonType("Cancel",
                                  ButtonBar.ButtonData.CANCEL_CLOSE));
Optional<ButtonType> result = alert.showAndWait();
// Decide what to do according to the button selected
if (result.get() == buttons[0])
       { System.out.println("That is so great to know!"); }
else if (result.get() == buttons[1]) {System.out.println("You make us happy."); }
else if (result.get() == buttons[2]) { System.out.println("We are glad you are
                                                                  pleased."); }
else if (result.get() == buttons[3]) { System.out.println("Uh oh ... sounds like
                                                      we need to improve."); }
else if (result.get() == buttons[4]) { System.out.println("Oh no! Please explain
                                                                       why."); }
```

- There is a limit to how many buttons would fit nicely on the window, but this gives you an idea.
- In addition to the **Alert** class, there is a **ChoiceDialog** class that allows the user to have a list of choices specified in a drop-down list as follows:

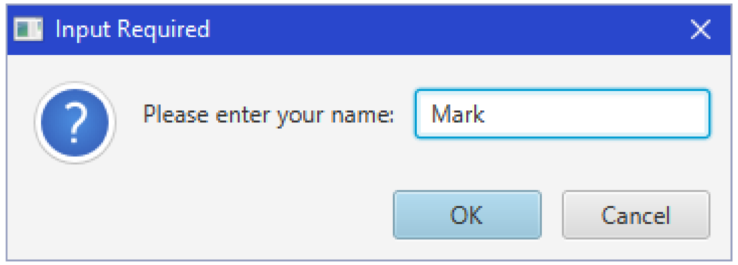




- To do this, we need to specify options again and set them in the constructor:

```
String[] options = {"Apple", "Orange", "Strawberry", "Banana", "Peaches"};
ChoiceDialog<String> dialog = new ChoiceDialog<String>("Peaches", options);
dialog.setTitle("Fruit Information"); dialog.setHeaderText(null);
dialog.setContentText("Choose your favorite fruit");

Optional<String> result = dialog.showAndWait();

if (result.isPresent()){

        System.out.println("Your choice: " + result.get()); }
```

- Notice that we can also set the default value (i.e., "Peaches" in this case) within the constructor.
- The **isPresent()** method allows us to determine whether or not the user selected an item or pressed CANCEL.
- If he/she did not press CANCEL, then we use the **get()** method to get the value that was selected.
- We can also create a **TextInputDialog** which will allow the user to enter text from a simple text field:



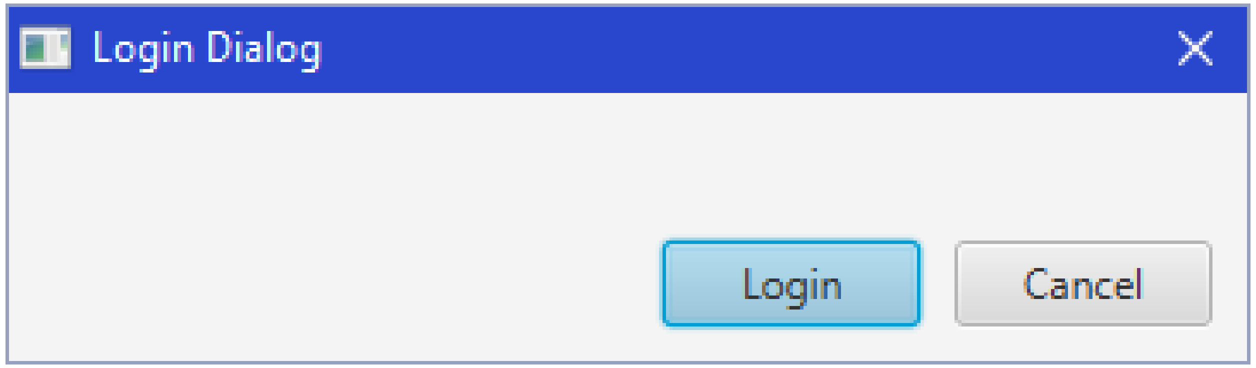
- The code to create the box is intuitive:

```
TextInputDialog dialog = new TextInputDialog("Mark");
dialog.setTitle("Input Required"); dialog.setHeaderText(null);
dialog.setContentText("Please enter your name:");
Optional<String> result = dialog.showAndWait();
if (result.isPresent()){
    System.out.println("Your name is: " + result.get()); }
```

As a final dialog box, let's customize one that will allow a Username and Password to be entered. To make a *general dialog box*, we use the **Dialog** class:

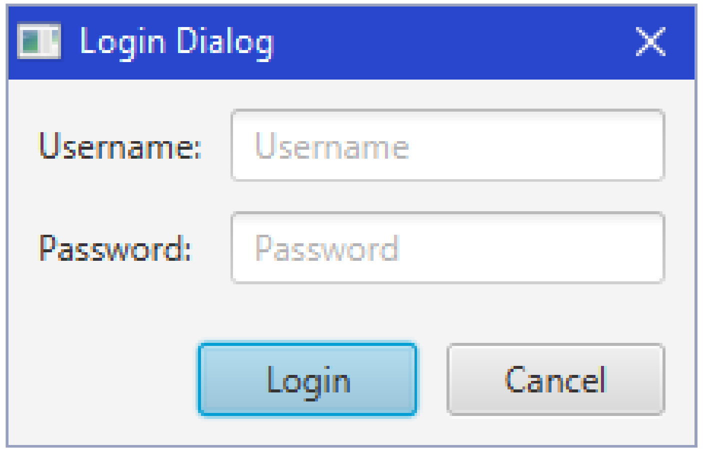
```java
Dialog dialog = new Dialog();
dialog.setTitle("Login Dialog");
dialog.setHeaderText(null);
ButtonType loginButtonType = new ButtonType("Login",
                                            ButtonBar.ButtonData.OK_DONE);
dialog.getDialogPane().getButtonTypes().addAll(loginButtonType,
                                                 ButtonType.CANCEL);
```

- This is a basic dialog box with a **Login** and **Cancel** button. You can change the "Login" to "Ok" for a more general dialog box.



- Now we can add a username and password field. We can create a **GridPane** to hold it all:
- Here is the code to do this:

```
GridPane grid = new GridPane();
grid.setHgap(10); grid.setVgap(10);
grid.setPadding(new Insets(10, 10, 10, 10));
TextField username = new TextField(); username.setPromptText("Username");
PasswordField password = new PasswordField();
password.setPromptText("Password");
grid.add(new Label("Username:"), 0, 0); grid.add(username, 1, 0);
grid.add(new Label("Password:"), 0, 1); grid.add(password, 1, 1);
dialog.getDialogPane().setContent(grid);
```



- Notice that there is a **PasswordField** object. This is just a **TextField**, but it hides the characters that the user types by replacing them with * characters for privacy.
- Finally, we can get open the dialog box and grab the results as follows:

```java
Optional result = dialog.showAndWait();
if (result.isPresent()) {
        System.out.println("Username = " + username.getText() + ", Password = "
                                                        + password.getText());
}
```

- This code accesses the username and password fields directly.
- This works because we are writing all of this code in one spot.
- However, if we created the dialog box elsewhere and we wanted to use it in various locations, we would not be able to access the username and password fields directly.
- In this case, we would need to adjust the return value of the Dialog box so that it contains the information that we need.

- Notice that there is a **PasswordField** object. This is just a **TextField**, but it hides the characters that the user types by replacing them with * characters for privacy.
- Finally, we can get open the dialog box and grab the results as follows:

```
Optional result = dialog.showAndWait();
if (result.isPresent()) {
    System.out.println("Username = " + username.getText() + ", Password = "
                                            + password.getText());
}
```

- This code accesses the username and password fields directly.
- This works because we are writing all of this code in one spot.
- However, if we created the dialog box elsewhere and we wanted to use it in various locations, we would not be able to access the username and password fields directly.
- In this case, we would need to adjust the return value of the Dialog box so that it contains the information that we need.

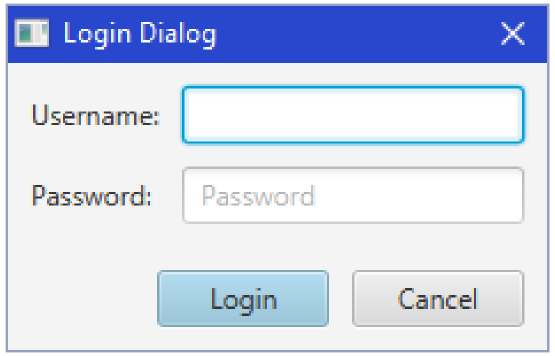
- We can replace the above code with this code:

```
// Convert the result to a Pair containing the username and password
dialog.setResultConverter( new Callback<ButtonType, Pair<String, String>>()
{
        public Pair<String, String> call(ButtonType b) {
                if (b == loginButtonType) { return new
                        Pair<String,String>(username.getText(),
                                                        password.getText());
                }
        return null;
        }
});
Optional<Pair<String, String>> result = dialog.showAndWait();
if (result.isPresent()) System.out.println("Username = " +
        result.get().getKey() + ", Password = " + result.get().getValue());
```

- Notice how we define a *Callback* (i.e., event handler) by using the **setResultConverter()** method.
- This allows us to define what will be returned.
- It has a return type of `Pair<String, String>`.
- This is simply a pair of strings.

- The code inside the callback sets the pair to be the username and password that was entered, as long as the LOGIN button was pressed.
- If CANCEL was pressed, or the window was closed, the result will be **null**.
- We then access this pair by calling **get()** and then we can use **getKey()** and **getValue()** to go inside the pair to get the username and password values.
- There are other minor adjustments that we can make. For example, we could have the dialog box come up with the focus in the username field:

```
username.requestFocus();
```



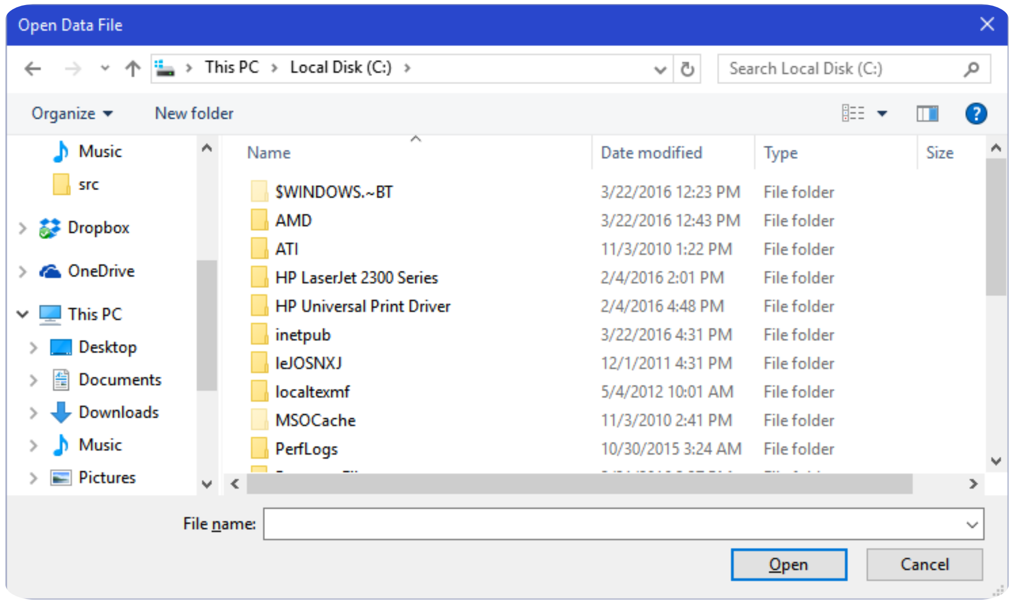
- This allows the user to start typing right away when the dialog box first opens instead of having to click on the username field to start typing. It is not necessary, but it is convenient.

- We can also disable the LOGIN field unless the user has typed in something:

```
        // Enable/Disable login btn depending on whether username was entered.
        Node loginButton = dialog.getDialogPane(). lookupButton(loginButtonType);
        loginButton.setDisable(true); // Disable upon start
        username.textProperty().addListener(new ChangeListener() {
                public void changed(ObservableValue observable, Object oldValue,
Object newValue) {
                        loginButton.setDisable(((String)newValue).trim().isEmpty());
                }
        });
```

- Now we have a nice username/password field dialog box. True, it is not a "standard" dialog box since we have customized it, but it is a type of dialog box that is commonly used.

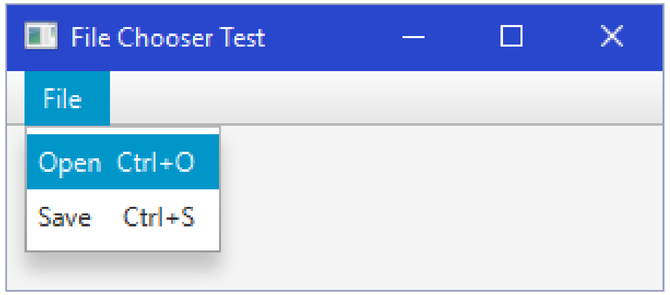
File Chooser Dialog Box



Name	Date modified	Type	Size
$WINDOWS.~BT	3/22/2016 12:23 PM	File folder	
AMD	3/22/2016 12:43 PM	File folder	
ATI	11/3/2010 1:22 PM	File folder	
HP LaserJet 2300 Series	2/4/2016 2:01 PM	File folder	
HP Universal Print Driver	2/4/2016 4:48 PM	File folder	
inetpub	3/22/2016 4:31 PM	File folder	
leJOSNXJ	12/1/2011 4:31 PM	File folder	
localtexmf	5/4/2012 10:01 AM	File folder	
MSOCache	11/3/2010 2:41 PM	File folder	
PerfLogs	10/30/2015 3:24 AM	File folder	

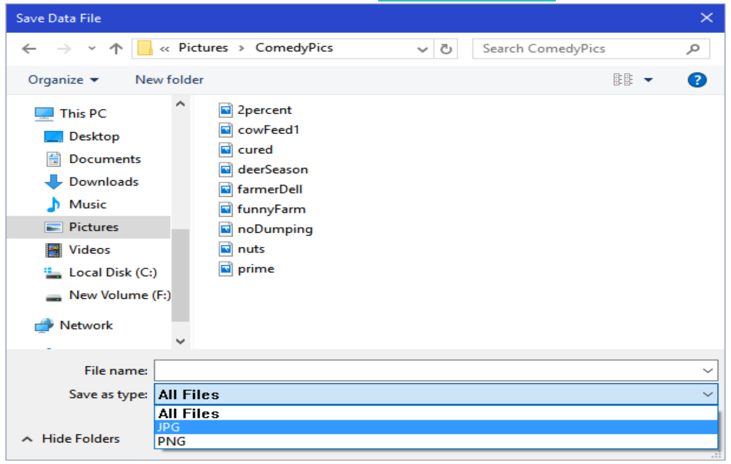
- There is another useful ***standard*** dialog box in JAVA that is used for selecting files. It is called a **FileChooser**.
- It allows you to browse around various directories and choose a file to open. A similar dialog box is used for choosing a file to save (in that case the difference is that the **Open** button is labelled as **Save** instead).

- The code brings up the appropriate **Open** or **Save** dialog box and then waits for the user to select a file name. It then displays the file name and the full path of the file.



```java
public class FileChooserTestProgram extends Application {
       private MenuItem openItem, saveItem;
       public void start(Stage primaryStage) {
       VBox p = new VBox();
       Scene scene = new Scene(p, 300, 100); // Set window size
       // Create the File menu
       Menu fileMenu = new Menu("_File");
       openItem = new MenuItem("Open");
       openItem.setAccelerator(KeyCombination.keyCombination("Ctrl+O"));
       saveItem = new MenuItem("Save");
       saveItem.setAccelerator(KeyCombination.keyCombination("Ctrl+S"));
       fileMenu.getItems().addAll(openItem, saveItem);
       openItem.setOnAction(new EventHandler<ActionEvent>() {
              public void handle(ActionEvent e) {
                     FileChooser chooser = new FileChooser();
                     chooser.setTitle("Open Data File");
                     File f = chooser.showOpenDialog(primaryStage);
                     if (f != null) {
                            System.out.println("File chosen to open: " + f.getName());
                            System.out.println("File with full path: " +
                                                                   f.getAbsolutePath());
                     }
              }
       });
```

```
        saveItem.setOnAction(new EventHandler<ActionEvent>() {
                public void handle(ActionEvent e) {
                        FileChooser chooser = new FileChooser();
                        chooser.setTitle("Save Data File");
                        File f = chooser.showSaveDialog(primaryStage);
                        if (f != null) {
                                System.out.println("File chosen to save as: " +
                                                                        f.getName());
                                System.out.println("File with full path: " +
                                                                f.getAbsolutePath());
                        }
                }
        });
        // Add the menu to a menubar and then add the menubar to the pane
        MenuBar menuBar = new MenuBar();
        menuBar.getMenus().addAll(fileMenu);
        p.getChildren().add(menuBar);
        primaryStage.setTitle("File Chooser Test");
        primaryStage.setScene(scene);
        primaryStage.show();
        }
        public static void main(String[] args) {
                launch(args);
        }
}
```



File name:

Save as type: All Files

- Notice that we simply open the "Open" file dialog box as follows:

```
        FileChooser chooser = new FileChooser();
        File f = chooser.showOpenDialog(primaryStage);
```

- We open the "Save" file dialog box similarly, but by using a different method name:

```
        FileChooser chooser = new FileChooser();
        File f = chooser.showSaveDialog(primaryStage);
```

- In either case, the dialog box returns a **File** object (more on this later).
- We can extract the file's name and its full path name by using **getName()** or **getAbsolutePath()**, which both return String objects.
- There are other settings that you can apply to the **FileChooser** dialog box.
- For example, you could have it open up in a particular directory by adding this before opening the chooser:

```
        chooser.setInitialDirectory(new File("C:\\"));
```

- You can insert any string representing a file folder/directory.
- You can even use **System.getProperty(...)** to get information from the system such as HOME-related system variables.
- You can also specify various file filters so that only certain types of files are shown:

```
        chooser.getExtensionFilters().addAll(
                    new FileChooser.ExtensionFilter("All Files", "*.*"),
                    new FileChooser.ExtensionFilter("JPG", "*.jpg"),
                    new FileChooser.ExtensionFilter("PNG", "*.png") );
```

- The above code allows either All Files, or just JPG and PNG files to be displayed or selected.