




Application Program Development

Segment : Crafting Your Own Dialog Boxes
Mahboob Ali



Outcomes

- Understanding how to
 - Create your own dialog boxes
 - Manage your own dialog boxes
 - Making and using different option with dialog boxes
- 

Making My Own Dialog Boxes

- Dialog boxes have an **owner** which is the window that caused it to appear.
- This allows the dialog box to be closed automatically when the user quits the application from the main window (i.e., all windows belonging to the same application are closed when the application shuts down). Also, when the owner window is minimized, the dialog boxes are also minimized.
- Normally, an application communicates with its dialog box through a **model** of some kind.
- That is, the owner opens up a dialog box, passing model-specific information to it.
- The user may then change this information from the dialog box, which in turn modifies the **model**.
- When the dialog box is closed, then the main application continues with the modified model objects.
- On next slide you will notice that the model is used as the "middle-man" between the two windows.
- That is, when the dialog box is first opened, the model contents are used to populate the components (i.e., fill in the text fields, button selections etc...).
- The user then makes appropriate changes to the components. When the dialog box is closed with the **OK** button, the model is updated with these new changes.
- When the dialog box is closed with the **CANCEL** button, the model remains unchanged.
- When either button is clicked, the dialog box closes.
- The closing of the dialog box using the standard "close" (i.e., X at the top corner) should be treated as a cancel operation.

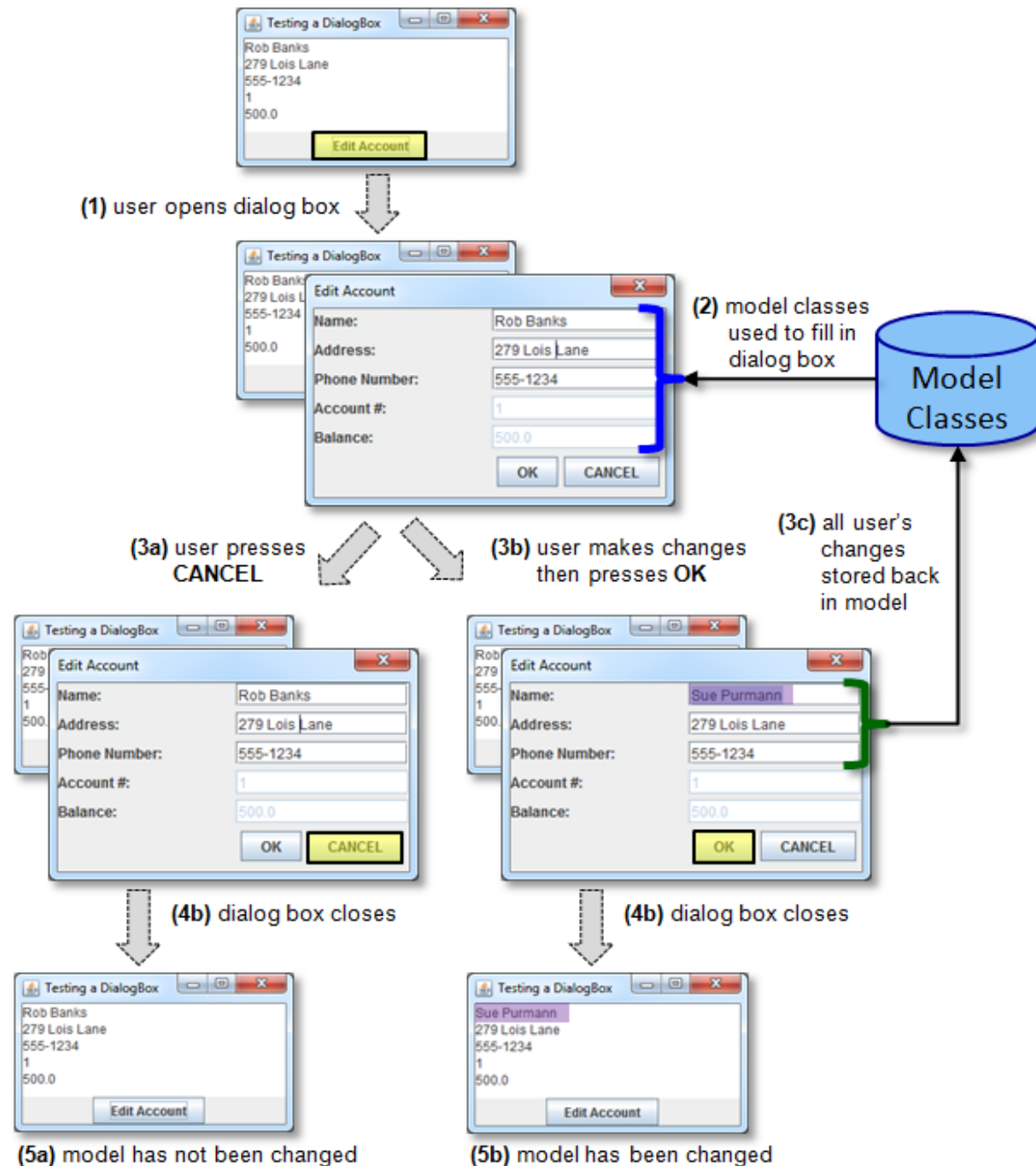
- The dialog box itself is easy to make. It is simply another window. To create your own, simply make it a subclass of **Dialog**:

```
public class MyDialogBox extends Dialog {  
    ...  
}
```

- Then, you can add components and event handlers to this dialog box as if it were a **Pane**.
- Typically, you will ensure that there is some combination of ok/apply and cancel/close buttons which are usually located at the bottom right or bottom center of a dialog box.
- There are various constructors in the **Dialog** class. We will use the following format for our constructors:

```
public MyDialog(Stage owner, String title, ...) {  
    setTitle(title);  
}
```

- The **owner** parameter is usually the main application's **Stage** object.
- The **title** parameter is what will appear on the dialog box title bar.
- We may also want to supply additional model-related parameters to pass information into the dialog box.
- Often the model itself is passed in as a parameter so that we can (1) fill in the dialog box information based on the current model data, and (2) then we can modify the model as necessary after the user makes changes to the data and presses OK.



- So, dialog boxes are easy to create ... but how do we coordinate the interaction with the main application window and its dialog box ?

- The dialog box is defined in a separate class than its owner application.
- As a result, the *owner* (i.e., the application that brought up the dialog box) has no idea what is going on within the dialog box class (nor should it need to know).
- The *owner*, however, usually needs to know whether or not the interaction with the dialog box was **accepted** (i.e., OK was pressed) or whether or not it was **cancelled**.
- That is, it may need to know whether or not changes were made to the model.
- The simplest way to do this is to define what is to be returned when the dialog box closes.
- The **Dialog** class has a method called **setResultConverter()** which allows us to provide a function to be called which will define what is to be returned from the **Dialog** box when it is closed.
- We can return **null** to indicate that it was cancelled, and something else otherwise. Here is the basic format:

```

setResultConverter(new Callback<ButtonType, RETURN_TYPE>() {
    public RETURN_TYPE call(ButtonType b) {
        if (b == okButtonType) {
            RETURN_TYPE result = new RETURN_TYPE();
            // ... Extract data from the Dialog box to fill in result ... return result;
        }
        return null;
    }
});

```

- Here, **RETURN_TYPE** could be any type of object that we want returned from the dialog box.




- When bringing up the **Dialog** box, we use this code:

```
Optional<RETURN_TYPE> result = myDialog.showAndWait();  
    if (result.isPresent()) { // Do something }  
    else { // Do something else }
```

- The *RETURN_TYPE* should match what was set in the dialog box.



EmailBuddy Example (Custom Dialog Box)

- Consider having many "buddies" (i.e., friends) that you send e-mails to regularly.
 - Consider making a nice little "electronic" address book that you can store the buddy's names along with his/her e-mail addresses.
 - Perhaps you even want to categorize the buddies as being "hot" (i.e., you talk to them often), or "not-so-hot". What exactly is an e-mail buddy ?
 - Well, we can easily develop a simple model of an **EmailBuddy** as follows
- 

Model Class for EmailBuddy

```
public class EmailBuddy {  
    private String name, address;  
    private boolean onHotList;  
    // Here are some constructors  
    public EmailBuddy() { name = ""; address = ""; onHotList = false; }  
    public EmailBuddy(String aName, String anAddress) {  
        name = aName; address = anAddress; onHotList = false; }  
    // Here are the get methods  
    public String getName() { return name; }  
    public String getAddress() { return address; }  
    public boolean onHotList() { return onHotList; }  
    // Here are the set methods  
    public void setName(String newName) { name = newName; }  
    public void setAddress(String newAddress) { address = newAddress; }  
    public void onHotList(boolean onList) { onHotList = onList; }  
    // The appearance of the buddy  
    public String toString() { return (name); }  
}
```

Model Class for EmailBuddyList

- As you may have noticed, there is nothing difficult here ... just your standard "run-of-the-mill" model class.
- However, this class alone does not represent the whole model for our GUI since we will have many of these **EmailBuddy** objects.
- So, we will need a class to represent the list.

```
public class EmailBuddyList {
    public final int MAXIMUM_SIZE = 100;
    private EmailBuddy[] buddies;
    private int size;
    public EmailBuddyList() {
        buddies = new EmailBuddy[MAXIMUM_SIZE];
        size = 0;
    }
    // Return the number of buddies in the whole list
    public int getSize() { return size; }
    // Return all the buddies
    public EmailBuddy[] getEmailBuddies() { return buddies; }
    // Get a particular buddy from the list, given the index
    public EmailBuddy getBuddy(int i) { return buddies[i]; }
    // Add an email buddy to the list unless it has reached its capacity
    public void add(EmailBuddy buddy) {
        // Make sure that we do not go past the limit
        if (size < MAXIMUM_SIZE) buddies[size++] = buddy;
    }
}
```

```

// Remove the buddy with the given index from the list
public void remove(int index) {
    // Make sure that the given index is valid
    if ((index >= 0) && (index < size)) {
        // Move every item after the deleted one up in the list
        for (int i=index; i<size-1; i++)
            buddies[i] = buddies[i+1];
        size--; // Reduce the list size by 1
    }
}

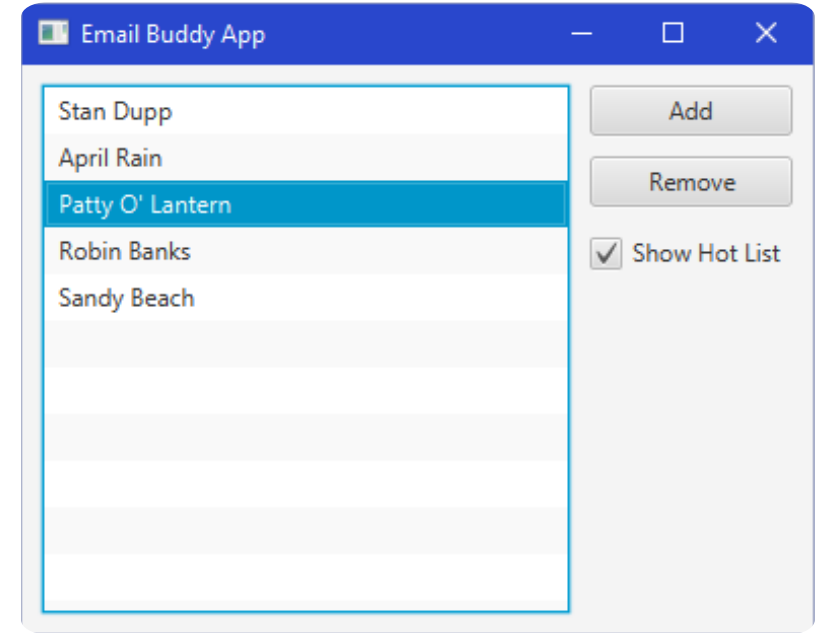
// Return the number of buddies on the hot list
public int getHotListSize() {
    int count = 0;
    for (int i=0; i<size; i++)
        if (buddies[i].onHotList()) count++;
    return count;
}

// Get a particular "hot" buddy from the list, given the hot list index
public EmailBuddy getHotListBuddy(int i) {
    int count = 0;
    for (int j=0; j<size; j++) {
        if (buddies[j].onHotList()) {
            if (count == i)
                return buddies[j];
            count++;
        }
    }
    return null;
}
}

```

- Notice that there is a **getSize()** method that is a simple "get" method and there is also a **getHotListSize()** method that returns the number of buddies on the hot list.
- Notice as well that there are methods to get a buddy at a given index in the array.
- The method **getHotListBuddy()** will find the **i**th buddy that is on the hot list. You will see soon why these methods will be useful.
- The task now is to design a nice interface for the main application.
- To start, we must decide what the interface should do. Here is a possible interface:
 - A **list** of all buddies is shown (names only)
 - We should be able to: o **Add** and **Remove** buddies from the list
 - **Edit** buddies when their name or email changes
 - Show only those buddies that are **"hot"** or perhaps show all of them

- Assume that we have decided upon the following view for the interface:
- Notice that the interface does not show the e-mail addresses in the list.
- It may look cluttered, but we could certainly have done this.
- Perhaps we could have made a second list box or something that would show the e-mail addresses.
- Here is a practice exercise:
 - make a **TextField** just beneath the list that will show the e-mail address of the currently selected **EmailBuddy** in the list. This is not hard to do.



- How can we build the view for this interface ? We will start by making a special **GridPane** and place various components on it:

```
public class EmailBuddyPanel extends GridPane {  
    private EmailBuddyList model; // This is the list of buddies  
    // The components on the window  
    private ListView<EmailBuddy> buddyList;  
    private Button addButton;  
    private Button removeButton;  
    private CheckBox hotListButton;  
    public ListView<EmailBuddy> getBuddyList() { return buddyList; }  
    public Button getAddButton() { return addButton; }  
    public Button getRemoveButton() { return removeButton; }  
    public CheckBox getHotListButton() { return hotListButton; }  
    public EmailBuddyPanel(EmailBuddyList m) {  
        model = m; // Store the model so that the update() method can access it  
        setPadding(new Insets(10, 10, 10, 10));  
        buddyList = new ListView<EmailBuddy>();  
        buddyList.setItems(FXCollections.observableArrayList(m.getEmailBuddies()));  
        add(buddyList, 0, 0, 1, 3); // spans 1 column, 3 rows  
        buddyList.setPrefHeight(Integer.MAX_VALUE);  
        buddyList.setMinWidth(200);  
        buddyList.setPrefWidth(Integer.MAX_VALUE);  
    }  
}
```

```
addButton = new Button("Add"); add(addButton, 1, 0);
setMargin(addButton, new Insets(0, 0, 10, 10));
setValignment(addButton, VPos.TOP); setHalignment(addButton, HPos.CENTER);
addButton.setMinHeight(25); addButton.setMinWidth(100);
removeButton = new Button("Remove"); add(removeButton, 1, 1);
setMargin(removeButton, new Insets(0, 0, 10, 10));
setValignment(removeButton, VPos.TOP);
setHalignment(removeButton, HPos.CENTER); removeButton.setMinHeight(25);
removeButton.setMinWidth(100);
hotListButton = new CheckBox("Show Hot List");
add(hotListButton, 1, 2);
setMargin(hotListButton, new Insets(0, 0, 10, 10));
setValignment(hotListButton, VPos.TOP);
setHalignment(hotListButton, HPos.CENTER);
hotListButton.setMinHeight(25);
hotListButton.setMinWidth(100);
// Now update the components by filling them in
update();
}
// Update the components so that they reflect the contents of the model
public void update() { //... coming soon ... }
}
```

- Notice that we are making our class a subclass of **GridPane**.
- That allow us to directly add the components by using **add()** within the constructor.
- Also notice that the constructor takes an **EmailBuddyList** as a parameter.
- We will fill in the list with the data from this parameter.
- The list is of type `ListView<EmailBuddy>` which allows us to populate it with **EmailBuddy** objects.
- At the end of the constructor, we call the **update()** method.
- Recall that the **update()** method should read from the model and then refresh the "look" of the components.
- The only components that need their appearance updated is the list and the remove button.
- The remove button is easily updated as we simply disable it when there is nothing selected in the list:

```
removeButton.setDisable(buddyList.getSelectionModel().getSelectedIndex() < 0);
```

- The list is more complicated. First of all, we need to populate the list with the most recent data.
- Recall that we did something similar in the grocery list example.

- We need to create an appropriate-sized array and then fill it up with email buddies and then set the list data

```
EmailBuddy[] exactList;  
exactList = new EmailBuddy[model.getSize()];  
for (int i=0; i<model.getSize(); i++)  
    exactList[i] = model.getBuddy(i);  
buddyList.setItems(FXCollections.observableArrayList(exactList));
```

- However, things are a little more difficult now. If we have the hot list button selected, then we do not want all the buddies ... instead we want only those on the hot list:

```
exactList = new EmailBuddy[model.getHotListSize()];  
for (int i=0; i<model.getHotListSize(); i++)  
    exactList[i] = model.getHotListBuddy(i);  
buddyList.setItems(FXCollections.observableArrayList(exactList));
```

- We can use an **IF** statement to select the appropriate code:

```
EmailBuddy[] exactList;  
if (hotListButton.isSelected()) {  
    exactList = new EmailBuddy[model.getHotListSize()];  
    for (int i = 0; i < model.getHotListSize(); i++)  
        exactList[i] = model.getHotListBuddy(i);  
}  
else {  
    exactList = new EmailBuddy[model.getSize()];  
    for (int i=0; i<model.getSize(); i++)  
        exactList[i] = model.getBuddy(i);  
}  
  
buddyList.setItems(FXCollections.observableArrayList(exactList));
```

- One last point ... as we will see later when editing a buddy, sometimes the **ListView** does not refresh properly. To fix this, we simply first need to set the **ListView** contents to **null** first before setting it to the value that we want.

```
buddyList.setItems(null); /// Seems to be required for a proper update  
buddyList.setItems(FXCollections.observableArrayList(exactList));
```

- We will also need to ensure that we select the selected item each time we make an update.
- That is, if we were to select an item from the list and then update ... we want to make sure that the item remains selected.
- At this point, when we refresh the list contents, the selected item does not remain selected.
- So, we will need to remember which item was selected and then reselect it again after the list is re-populated.
- Here is the final **update()** method that must be added to the view code:

```
// Update the components so that they reflect the contents of the model  
public void update() {  
    // Remember what was selected  
    int selectedItem = buddyList.getSelectionModel().getSelectedIndex();  
    // Now re-populate the list by creating and returning a new  
    // array with the exact size of the number of items in it.  
    EmailBuddy[] exactList;  
    if (hotListButton.isSelected()) {  
        exactList = new EmailBuddy[model.getHotListSize()];  
        for (int i = 0; i < model.getHotListSize(); i++)  
            exactList[i] = model.getHotListBuddy(i);  
    }  
}
```

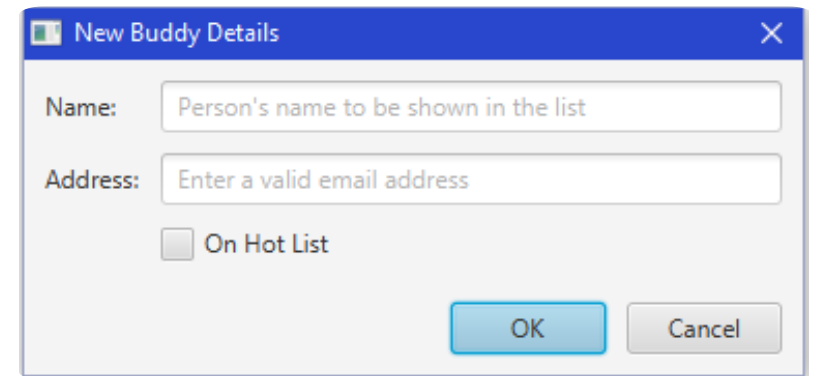
```
else {  
    exactList = new EmailBuddy[model.getSize()];  
    for (int i=0; i<model.getSize(); i++)  
        exactList[i] = model.getBuddy(i);  
}  
  
buddyList.setItems(null); /// Seems to be required for a proper update  
buddyList.setItems(FXCollections.observableArrayList(exactList));  
  
/// Reselect the selected item  
buddyList.getSelectionModel().select(selectedItem);  
  
/// Enable/disable the Remove button accordingly  
removeButton.setDisable(buddyList.getSelectionModel().getSelectedIndex() < 0);  
}
```

- The **controller** will keep track of the **view** as well as the **model**.
- We will be handling events for the pressing of the **addButton**, **removeButton**, **hotListButton** as well as **buddyList** selection.
- Here is the basic framework for the controller:

```
public class EmailBuddyApp extends Application {  
    private EmailBuddyList model; // The model  
    private EmailBuddyPanel view; // The view  
    public void start(Stage primaryStage) {  
        // Initially, no buddies  
        model = new EmailBuddyList();  
        // Make a new viewing panel and add it to the pane  
        view = new EmailBuddyPanel(model);  
        //Handle the Add button  
        view.getAddButton().setOnAction(new EventHandler<ActionEvent>() {  
            // This is the single event handler for all of the buttons  
            public void handle(ActionEvent actionEvent) {  
                // Add buddy (code will be shown later) }  
            });  
        // Handle the Remove button  
        view.getRemoveButton().setOnAction(new EventHandler<ActionEvent>() {  
            // This is the single event handler for all of the buttons  
            public void handle(ActionEvent actionEvent) {  
                // Remove buddy (code will be shown later) }  
            });  
    }  
}
```

```
// Handle the Hot List Button
view.getHotListButton().setOnAction(new EventHandler<ActionEvent>() {
    // This is the single event handler for all of the buttons
    public void handle(ActionEvent actionEvent) { view.update(); }
});
// Handle a double-click in the list
view.getBuddyList().setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent mouseEvent) {
        if (mouseEvent.getClickCount() == 2)
            // Edit buddy (code will be shown later)
            view.update();
    }
});
primaryStage.setTitle("Email Buddy App");
primaryStage.setScene(new Scene(view, 400, 300));
primaryStage.show();
}
public static void main(String[] args) { launch(args); } }
```

- Notice that the code is straight forward.
- The hot list button event handler only requires a refreshing of the list, so the view's **update()** method is called.
- We now need to decide what to do when the user clicks the **Add** button, **Remove** button and **OnHotList** button as well as when the user selects an item from the list.
- The **Add** button should bring up a dialog box to allow us to add a new buddy.
- We have not created this dialog box, but we will do so soon.
- The adding of the new email buddy should only occur if the user presses the **OK** button.
- If the **CANCEL** button is pressed, or the dialog box is closed down, then no email buddy should be added.
- To make the code simpler, it is a good idea to create the new email buddy when the **Add** button is pressed so that we can pass this buddy into the dialog box so that its contents can be set.
- If the user presses **OK** afterwards, we can add this new buddy to the model.
- The dialog box should allow the user to set the name, address and hot list status for the buddy that it is working on (i.e., either a newly added buddy or one being edited). Here is what the dialog box will look like:



The dialog box is titled "New Buddy Details" and contains the following elements:

- Name:** A text input field with the placeholder text "Person's name to be shown in the list".
- Address:** A text input field with the placeholder text "Enter a valid email address".
- On Hot List:** A checkbox that is currently unchecked.
- Buttons:** "OK" and "Cancel" buttons at the bottom right.

- We can create this dialog in a class called **BuddyDetailsDialog**.
- It will be a subclass of the **Dialog** class.
- The constructor will take three parameters:
 - the owner, which is a stage
 - a title for the window
 - an **EmailBuddy** to be edited.
- Here is the basic code that will bring up the window with the two **Labels**, **TextFields** and **Checkbox**:

```
public class BuddyDetailsDialog extends Dialog {
    public BuddyDetailsDialog(Stage owner, String title, EmailBuddy bud) {
        setTitle(title);
        // Set the button types
        ButtonType okButtonType = new ButtonType("OK",
            ButtonBar.ButtonData.OK_DONE);
        getDialogPane().getButtonTypes().addAll(okButtonType,
                                                    ButtonType.CANCEL);

        // Create the and and address labels and fields.
        GridPane grid = new GridPane(); grid.setHgap(10); grid.setVgap(10);
        grid.setPadding(new Insets(10, 10, 10, 10));
        TextField nameField = new TextField(bud.getName());
        nameField.setPromptText("Person's name to be shown in the list");
        nameField.setMinWidth(300);
```



```
TextField addressField = new TextField(bud.getAddress());
addressField.setPromptText("Enter a valid email address");
addressField.setMinWidth(300);

CheckBox onHotList = new CheckBox("On Hot List");
onHotList.setSelected(bud.onHotList()); grid.add(new Label("Name:"), 0, 0);
grid.add(nameField, 1, 0); grid.add(new Label("Address:"), 0, 1);
grid.add(addressField, 1, 1); grid.add(onHotList, 1, 2);
getDialogPane().setContent(grid); // Puts the stuff on the window
}

}
```

- The code is straight forward. However, there are some interesting points.
- Notice how the incoming **EmailBuddy bud** is used to populate the **TextFields** and set the **HotList** value.
- This allows the dialog box to come up with information already in the fields.
- Also notice how the **GridPane** is set for the dialog box in the last line.
- When the **OK** button is pressed, we need to update the **EmailBuddy** that was passed in as a parameter to have the name, address and hotList status as specified in the dialog box data.

- We write the **setResultConverter()** method which allows us to specify what to return to the main program that brought up this dialog box.
- In our case, we would like to return the **EmailBuddy** whose information was just added or edited.
- However, if the user pressed **CANCEL** or closed the window, then we do not want to return the **EmailBuddy**, but will return **null** instead ... to indicate that nothing is to be changed.

```
// Convert the result to an EmailBuddy containing the info
setResultConverter(new Callback<ButtonType, EmailBuddy>() {
    public EmailBuddy call(ButtonType b) {
        if (b == okButtonType) {
            bud.setName(nameField.getText());
            bud.setAddress(addressField.getText());
            bud.onHotList(onHotList.isSelected());
            return bud;
        }
        return null;
    }
});
```

- Notice that it checks the button type to see if it is the **okButtonType** that was created earlier.
- If so, it then extracts all the data from the dialog box and sets it for the buddy and returns the buddy. Otherwise, it returns **null**.
- The final addition that we will make is to disable the **OK** button unless the user has typed in both a name and an address ... or if the user selects/deselects the **onHotList** checkbox.
- The idea is to disable the button until something valid has been entered.
- To do this, we need event handlers to be called when the user types into a text field or checks the **Checkbox**. To start, we assume that the button should be disabled:

```
// Enable/Disable OK button depending on whether a username was entered.
```

```
Node okButton = getDialogPane().lookupButton(okButtonType);  
okButton.setDisable(true); // Disable upon start
```

- Then, the idea is to simply check to see whether or not both text fields have something in them.
- If they both have something, then we can enable the **OK** button, otherwise disable it. Here is the code to check that:

```
okButton.setDisable(nameField.getText().trim().isEmpty() ||  
                    addressField.getText().trim().isEmpty());
```

- Now we are ready to go back to the main application and handle the pressing of the **Add** button.
- In this case, we need to create a new **EmailBuddy**, and then bring up the **BuddyDetailsDialog** in order to allow the user to enter the data for that buddy.
- Here is the code for adding a buddy. It needs to be inserted in the code above as the **Add** button event handler:

```
public void handle(ActionEvent actionEvent) {
    EmailBuddy aBuddy = new EmailBuddy();
    // Now bring up the dialog box
    Dialog dialog = new BuddyDetailsDialog(primaryStage,
                                           "New Buddy Details", aBuddy);
    Optional<EmailBuddy> result = dialog.showAndWait();
    if (result.isPresent()) {
        model.add(aBuddy);
        // Add the buddy to the model
        view.update();
    }
}
```

- The code is easy to follow.
- Notice how we decide what to do depending on the result coming back from the showing of the dialog box.
- If the result was present (i.e., OK was pressed), then we simply add the newly created **EmailBuddy** and update the view to reflect the changes.
- Otherwise, if **CANCEL** was pressed, then there is nothing to do.

- For the **Remove** button, we simply need to look at what is selected from the **ListView** and remove it from the model:

```
public void handle(ActionEvent actionEvent) {  
    int index = view.getBuddyList().getSelectionModel().getSelectedIndex();  
    if (index >= 0) {  
        model.remove(index);  
        view.update();  
    }  
}
```

- Notice how the code simply determines the index of the selected item in the list and then calls the **remove()** method in the model in order to remove the item from the list.
- The call to **update()** simply refreshes the window.

- Finally, in order to be able to edit an **EmailBuddy**, we need to create an event handler for the **ListView**. If a single-click was done on the list, there is not much to do except call **update()** so that the **Remove** button could be enabled, allowing us to remove the selected item.
- Notice how we can ask the **MouseEvent** for the click count. Now, to perform the editing, we will need to determine the selected **EmailBuddy** and then open up the **BuddyDetailsDialog** with that buddy so that it can be edited. Here is the code:

```
public void handle(MouseEvent mouseEvent) {
    if (mouseEvent.getClickCount() == 2) {
        EmailBuddy selectedBuddy;
        int selectedIndex = view.getBuddyList().getSelectionModel().
                                                                    getSelectedIndex();

        if (selectedIndex >= 0) {
            if (view.getHotListButton().isSelected()) selectedBuddy =
                                                                    model.getHotListBuddy(selectedIndex);
            else selectedBuddy = model.getBuddy(selectedIndex);
            if (selectedBuddy == null) return;
            // Now bring up the dialog box
            Dialog dialog = new BuddyDetailsDialog(primaryStage, "Edit Buddy
                                                                    Details", selectedBuddy);

            Optional<EmailBuddy> result = dialog.showAndWait();
            if (result.isPresent()) {
                view.update();
            }
        }
    }
    else {
        view.update(); // Allows Remove button to be enabled on single click
    }
}
```

- Notice a couple of things.
 - First, we need to ensure that there is a selected item in the list, otherwise we do nothing. We just check to make sure that it is not -1.
 - Then, we need to determine which **EmailBuddy** was selected.
 - This will depend on whether or not the Hot List is being shown.
 - We have two model methods that we can call: **getBuddy()** and **getHotListBuddy()**.
- Each takes an integer indicating the number of the item in the list that we want.
- So, we just call the appropriate method and it will return the selected **EmailBuddy** object accordingly.
- If the result is **null**, then there is nothing to do. Otherwise, we need to edit.
- To edit, we simply open up the dialog box as before, but now with a different title and with the **selectedBuddy** instead of a newly created one.
- If the **OK** button was pressed, we simply update the list, otherwise there is nothing to do.
- How does the **EmailBuddy** get edited ?
 - Well, the code in the **setResultConverter()** of the **BuddyDetailsDialog** will ensure that if **OK** was pressed, then this **selectedBuddy**'s contents will be changed to that which is in the **TextFields**.
- It all works out rather smoothly.