

# Applying Forms Programming

(60 points, estimated time: 3 hrs)

## Overview

This assignment has you apply material from Callahan chapters 2-4 and the associated WebCasts. You'll develop the form depicted below, which presents employee information for MegaTelCo, a corporate communications provider. Notice the form contains a navigation combo box, a filtering option group, and a few command buttons. Users will be able to view and filter existing data but will be *unable* to edit records or delete records unless they use the buttons at the bottom of the form.

The screenshot shows an Access form window titled "Employees". At the top, there is a dropdown menu labeled "Employee to Display" with "Trollen, Tom" selected. Below this, the form displays the following fields and controls:

- EmployeeID: 1 (text box)
- Last Name: Trollen (text box)
- First Name: Tom (text box)
- Wage Rate: \$40.00 (text box)
- DOB: 8/16/1957 (text box)
- Hire Date: 1/15/1983 (text box)
- Is Retired:  (checkbox)
- Modified Date: (text box)
- Modified By: (text box)
- Employee Filter: A group box containing two radio buttons: "Active Employees" (unselected) and "All Employees" (selected).

At the bottom of the form, there are four buttons: "Edit Record", "Save Record", "Add Record", and "Delete Record". Below the buttons is a status bar showing "Record: 1 of 11", "Unfiltered", and a "Search" field.

## Steps

1. Use the link embedded in the Blackboard assignment to download **Applying Forms Programming.mdb** to your **CIS217 Data Files\Others** folder. Then rename the file to include your own name.
2. Open the database file. Notice it contains a table, a form, and a standard module.
3. Open *frmEmployee* in Form view. Since users will not interact with the *EmployeeID*, *ModifiedDate* and *ModifiedBy* fields, set their Enabled property to No, Locked to Yes, and BackStyle to Transparent.
4. Save your form and test these controls before continuing. They should resemble the illustration above and should not allow a user to click inside and make a change.
5. Set the Name of every non-label control to conform to the Balter naming convention.
6. Build code in an appropriate event handler to set form properties so the form initially does not allow edits, and doesn't allow deletions. Do not restrict the form's ability to add new records. [Do not set these properties in the Properties sheet.]
7. Switch into Form view and test these form properties. Try to edit an existing record. It should fail. Then try to delete a record... it should be grayed-out. If not, go back and modify your event handler code to set appropriate form properties to disallow these functions.

Next you'll develop the navigation combo box that makes it easier to view a different record.

8. Switch into Design View and check that Control Wizard is on. Then add a navigation combo box in the form header. Have the combo box rows display the employee's full name (*Last Name, First Name*, as illustrated), sorted in alphabetical order. Assign the combo box control an appropriate name. *Be careful*: when you change the Name of a control, any *existing* code that handles the control's events must be changed to reflect the control's new name.
9. Test your combo box by using it to navigate to other records. Does it work? If not, edit your code or delete the control and rebuild it, then test again.
10. Develop and test code that *synchronizes* the combo box whenever the user navigates to another record on the form. Your combo box must remain synchronized with the record displayed on the form.

Next you'll develop the *Edit Record* button.

11. Turn Control Wizards **off** and build your *Edit Record* button. After placing the button on the form, assign it an appropriate name and hotkey to match the illustration. Then build code that allows the record to be edited when a user clicks this button. Test your *Edit Record* button and modify a field. Then use Undo to abandon the changes.
12. You also need to build code in other events that returns the form to its read-only state when the user navigates to another record and after a record has been saved. Your form must not allow an edit of any other record.
13. Test your *Edit Record* button by modifying a record. Then use [Shift]+[Enter] to save the changes. Check that the form returns to its read-only state. Now, click the *Edit Record* button but do not make any changes, then navigate to another record. You must not be able to edit the record you just navigated to.

Now you'll develop the *Save Record* button.

14. Turn Control Wizards back **on** and build your *Save Record* button. When the user clicks this button, your code will save the current record. Be sure it has the hotkey specified in the illustration and bears a name that conforms to Balter's naming convention.
15. Test your *Save Record* button. Modify a record, then click the *Save Record* button. The pencil symbol should vanish (the record was saved!). If not, edit your code and test again before continuing.
16. After a record is saved, your form should display a message confirming that the employee record was saved and should return the form to its read-only state. Recall that there are several ways a user could save a record and that one particular event occurs after any successful save. Consider which event to use, and then develop your code.
17. Test your new code. Modify a record, then click *Save Record*. You should see the message indicating the employee record was saved. If not, edit your code and test again before continuing.
18. Now, close your form and reopen it. Although the record has not been changed, go ahead and click your *Save Record* button. You'll see a message indicating that 'Save Record' is not available now (because it has not been modified!). To prevent users from doing this, modify your form's code to disable the *Save Record* button when the form first appears by setting the button's Enabled property to False. Close and reopen the form to verify that users cannot click the *Save Record* button when the form first appears.
19. Now add code that enables the *Save Record* button as soon as data in the current record is modified. Also add code so that after a record has been saved, the *Edit Record* button receives the focus and the *Save Record* button becomes disabled. Also add code that disables the *Save Record* when a user navigates to another record or abandons changes to the record. Test that the *SaveRecord* button behaves as requested under each of these 4 situations.

Next, you'll build and test the *Add Record* button.

20. With the Control Wizards still on, build your *Add Record* button. Be sure it has the hotkey specified in the illustration and bears a name that conforms to Balter's naming convention.
21. Test your *Add Record* button. Does it work correctly? If not, delete it and test again.

Now that your *Add Record* button is functional, you'll add a refinement related to newly added records.

22. Use your form to add another employee, marking them as a *retired* employee. Save the record. Now notice your navigation combo box. The new employee's name is not displayed in the text box portion and does not appear among the rows of the list box portion. Modify the form's code module so the combo box does show newly added employees whenever a new one is added.
23. Add another new employee and test that this refinement works correctly. Modify and re-test your code so the combo box works as described when a new employee is added.
24. Close *Access* and backup your file before continuing.

Next, you'll build and test your *Delete Record* button

25. With the Control Wizards still on, build your *Delete Record* button. Be sure it has the hotkey specified in the illustration and bears a name that conforms to Balter's naming convention.
26. Once the button has been created, view the code for its Click event. Notice the statements:

```
DoCmd.RunCommand acCmdSelectRecord  
DoCmd.RunCommand acCmdDeleteRecord
```

The *RunCommand* method is used to execute Ribbon commands from within a VBA procedure. The first statement selects the current record. The second statement deletes the selected record.

27. Remain in the *Delete Record* button's Click event handler. Recall that you set a form property that *prevents* deletions. You need to add a line of code to this event handler that changes the form to *allow* deletions when the button is clicked. Enter such a line now.
28. Test the functionality of your *Delete Record* button. Attempt to delete one of the new employees you have added. Notice you get a warning dialog box indicating "*You're about to delete 1 record(s).*" Click *Yes*. Does your *Delete Record* button work correctly? If not, edit your code and test again.
29. Navigate to another record and click your *Delete Record* button. This time click *No* when the warning appears. You'll see a message that says "*The RunCommand action was cancelled*", indicating that the deletion was cancelled. (We'll see how to suppress this in a later chapter webcast.) Click *Ok* to close the message. The record remains, but a problem exists. Your form is still in the "allow deletions" state! To confirm this, check the *Home* tab's *Records* area, and pull down the *Delete* list box. Notice that *Delete Record* is still available (don't click it). Press *Esc* to close the *Delete* list box. Now navigate to another record and check again. *Delete Record* is still available!

You need to develop code that makes the form revert to its "no deletions" state after the user has chosen whether or not to delete the record. In either case, the form must not allow subsequent deletions until the *Delete Record* button is clicked again.

30. Actually, three events can occur when a user tries to delete a record. Use the Event Demonstrator application and/or Access Help to discover these events and their sequence. Select the one that occurs after a user has decided whether or not to delete the record and can detect whether or not the user cancelled the deletion. [DO NOT USE FORM\_CURRENT since it is called automatically as Access processes a record deletion.]
31. Now that you've identified the appropriate event, build code that makes the form revert to its "no deletions" state once the user has decided whether or not to delete the record. In either case, the form should not allow subsequent deletions until the *Delete Record* button is clicked again. To confirm, check the *Home* tab's *Records* area, and pull down the *Delete* list box.
32. Test your new code by deleting a record, then navigate to another and try to delete that record. Then, start to delete a record but answer *No* when asked to confirm. Check the *Delete* list box. Your form must not allow this deletion. If necessary, modify your code and test again.
33. As a courtesy to the user, modify this event handler's code to detect whether or not the user cancelled the deletion, then display a confirmation message (either "Employee retained." or "Employee deleted.").

34. Test your new code. First add a new employee named Bob Along, and save him. Then use your *Delete Record* button to delete him. Do you see your “Employee deleted.” message?

One last refinement relating to deleting records.

35. Click the combo box and look for Along, Bob. You should see **#Deleted** where his row used to be. Clearly the combo box needs to be updated after an employee is actually deleted! Select an appropriate event and use VBA code to update the combo box after a record is deleted.

36. Test your code by deleting a record. Does the combo box respond appropriately?

Next, you’ll build and test the filtering option group. The form is to display all employees until a user clicks the option to display only active (non-retired) employees.

37. Add the filtering option group, as illustrated. Give the option group frame an appropriate name and value. Then build the code that does appropriate filtering in response to user action in the option group.

38. Test each filtering option button to be sure they work correctly.

Recall that there are several ways a user could save a record and that there is an event that occurs regardless of which way the user saved the record. Just *before* a new or modified record is saved, *your code* should set the record’s *ModifiedDate* text box to the current date/time and set the *ModifiedBy* text box to the logged in person’s username. (Note: by setting the values for the *ModifiedDate* and *ModifiedBy* **text boxes**, they will be immediately visible on the form.)

39. Just before a record is saved, your form’s code should set that record’s *ModifiedDate* equal to the current date. Build an event handler that sets the *ModifiedDate* text box to the current date and time.

40. Test that your event handler works correctly. Edit a record, and then save the record. Does the current date/time appear in the *ModifiedDate* text box? If not, modify your event handler and test again.

41. Your event handler should also set the record’s *ModifiedBy* text box to the user’s username. This database has a code module named *basTrackActivity* that contains a function named *OSUser*. To become familiar with this function, press [Ctrl]+[G] to go to the Immediate Window, then type the expression **?OSUser()**. Your username should be returned as a character string.

42. Add code to your event handler that sets the *ModifiedBy* text box to the result of the *OSUser()* function.

43. Test that your event handler works as required. Modify and then save a record. Does your username now appear in the *ModifiedBy* text box? If not, modify your code and test again.

Wrap Up

44. Compact your file.

45. Close *Access* and backup your file.

46. Upload your file for grading.