X414.20 Fundamentals of Software Development

Coding Lab 9 – VB Menus, Subroutines and Functions

Please complete these lab steps only while in the Zoom meeting.

It is easiest to print these instructions so that you can refer to them during the lab. Your screen will be busy enough with both Zoom window and your Amazon Workspace window.

Task 1 – Add Menus to the Program (if you haven't done this last week)

1. Keith will give a demo of how to create menus.
2. Now it's your turn. Create a menu strip that contains: File Scenario Help
3. The File menu will have:
    a. Print Form (then make this disabled)
    b. A separator line
    c. Exit (this will exit the program)
4. The Scenario menu will have:
    a. Grad Student (this will make sure Student is checked and the hall is Saxon Suites)
    b. Faculty RA (this will make sure Student is not checked and Resident is checked)
5. The Help menu will have:
    a. About WelcomeUCLA… (This will display a copyright message)
6. Now that you have created the menu structure, each of these menu items (not the ones on the strip itself at the top) should have a Click event and in that you will write the code necessary to do what is indicated in the parentheses. **Do this now for all the items above.**
7. Test this out. When done, File / Save All and signal Keith to check this.

Task 2 – Increase Modularity through Code Subroutines

In this task, we illustrate how to create regular subroutines (as opposed to event subroutines) that can be called from your code. I call these ***code subroutines***, although the world usually just refers to these as subroutines.

We are going to create just a couple code subroutines, and we will move some of the existing code to these. I want you to see the mechanics of these subroutines, and then in the future, you will start designing your code with subroutines in mind.

Right now, you have some code in an **if** statement that enables (or makes visible) all the student-related controls: enabling the Major label and combo box, and making visible the label that displays the enrollment room. You also have code that disables (or makes invisible) the same. Let's take each of these sets of code and place them in code subroutines instead, so that the **if** statement only needs to call the appropriate subroutine.

We will create the structure of the subroutine definition first, which makes typing the actual call much easier.

1. Let's create the subroutine skeleton for a new subroutine called **EnableStudentStuff**. Higher up in the code—after the dim statements—place the cursor above the first event subroutine, and type (type it just as you see it below, mirroring the upper and lower case letters):

   ```
   private sub EnableStudentStuff
   ```

   and press ENTER. You'll notice that this automatically creates the skeleton, complete with an End Sub statement as well as adding **()** to the end of **EnableStudentStuff**. In VB, just as in C, subroutines are always expressed with their parenthesis, even if there are no arguments. It also capitalizes the Private and the Sub, because these are keywords. Your code should now look like:

   ```
   Private Sub EnableStudentStuff()

   End Sub
   ```

2. The code for this new subroutine goes into between these two lines. Let's move the "enabling" statements from the event subroutine **chkStudent_CheckedChanged** to this new **EnableStudentStuff()**. Highlight the three statements in the "Then" part of the **if** statement. Then type Ctrl+X to cut them (or right-click and choose Cut). Finally, paste them in between the **Private Sub EnableStudentStuff()** and **End Sub** lines.
3. Now that the definition of EnableStudentStuff() is finished, we can add a statement to the "Then" part of the **if** statement to call it. Type the following statement in the now empty "Then" part:

   ```
   EnableStudentStuff()
   ```

   and press ENTER. But….you'll notice as you start to type it, VB actually suggests it! That's because we defined it up at the top of the program. So if you don't want to type the whole thing, just start typing until you see the suggestion highlighted and then press TAB and ENTER.
4. Now test this out by running the program and making sure that checking and unchecking the Student checkbox still does what it is supposed to.
5. OK. Now that you've tackled this, let's do **DisableStudentStuff**. Go ahead and define the skeleton, then cut and paste the code from the "Else" part of the **if** statement to it. And finish up by typing the call in the "Else" part of the **if** statement.
6. Signal Keith when you get to this point so that he can review your code.

Task 3 – Create a Function (this is NOT use for Assignment 6)

Keith will give a demonstration of how functions work. You know how to call them already, and we did a little work in Module 4 on them. Now you will learn how to create a definition of a function in VB.

When we call a function, we give it an argument (or two or three or more). The idea is that the values from the arguments get "passed" to the inside of the function to be acted upon. The person calling the function need only to indicate the arguments being passed, in a certain order. It's like a customer giving a contractor instructions on a job:

> *here's my budget (a number),*
> *here's my completion date (a date),*
> *here's my style code (an alphabetic code).*
>
> *Now go build the thing I want, and hand it back to me completed (a completed storage cabinet)!*

The things the customer gives to the contractor are *arguments*.

On the contractor side, she is expecting the three arguments to be given to here. She accepts them, but she refers to them as the *parameters* for the job. It's what she needs to get started. She will then need nothing else from the customer. She works on the project with her own methods and people and resource, and then produces a finished product.

We do the same with functions. When you call a function, you provide the arguments. And the function definition says what you are going to do with those arguments and what you are going to product. Inside the function definition, we have special variables (created the coder who does the function) that accept the arguments and store them while inside the function. These variables are the *parameters*.

If there are, say, 3 arguments, then there should be 3 parameters. The first goes with the first, the second with the second and the third with the third. The data types of each argument-parameter pair need to match.

Inside the function, you will never use any of the variables from the code outside the function. You will ONLY use the parameters and any variables you have defined locally. Again, it's like the contractor using her own tools.

Inside the function, do all the necessary calculations and manipulations to arrive at the final answer. And then when you have that final answer, you hand that back to the call so that it can use the answer in an expression.

Let's call a function in cboMajor_SelectedIndexChanged to replace all the code that's there. The function call will look like this:

```
lblEnrollRoom.text = SelectRoom(cboMajor.Text)
```

SelectRoom() will be the function. It will have one argument, the string that's inside of **cboMajor.Text**. That will be passed into **SelectRoom().** You can call the parameter whatever you'd like; think about what data type you will use.