# Chapter 5

**Menus, Common Dialog Controls,**

**Context Menus, Sub Procedures, and Functions**

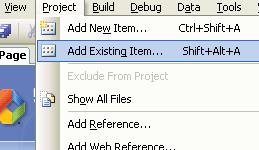
**Coverage:**

This chapter covers several important topics: (1) use of menus and context menus in place of and in addition to button controls, (2) writing general sub procedures and functions to modularize your code and make the logic easier to understand, and (3) the use of common dialog boxes – controls already designed to support common tasks such as printing, changing the interface (fonts and colors), and saving files.

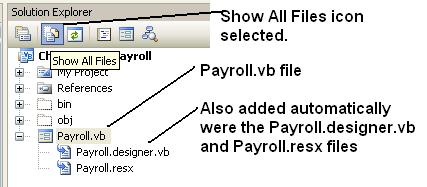
* Start VB and create a new project named **Ch05Payroll**.
* You will not be using the default **Form1** – instead you will add the form from the **Ch04VBUniversity-Solution** project to this project.
* Right-click the **Form1.vb** in the Solution Explorer window and select the **Delete** option – respond **Yes** to the question about **Form1** being permanently deleted.



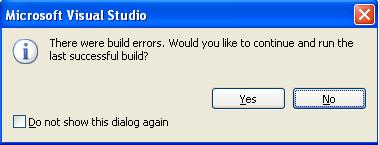
* To add the form from the Chapter04 project, select the **Project** **menu**, **Add** **Existing** Item option – browse to the **Ch04VBUniversity-Solution** folder and select the **Payroll.vb** file – this is the form file from that project.



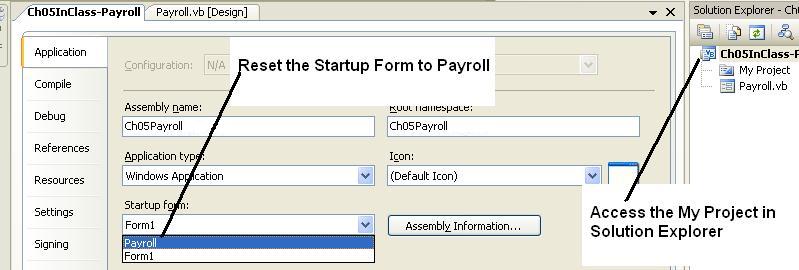
* You should see that the **Payroll.vb** file now displays in the Solution Explorer window—if the form is not open (displayed), click it to open the form. Click the **Show All Files** icon and expand the **Payroll.vb** link and you'll see that other files that are related to the **Payroll.vb** file were also imported – files storing the resources for this form.



* If you try to run the project now, you will get a build error like the one shown in this figure – the project is looking for **Form1** and it is no longer available. You need to reset the **Startup form property** for the project.



* Access the **My Project** file by double-clicking it in the Solution Explorer window. This displays properties of the project. Set the Startup form property to the **Payroll** form. Click the **Save All** button on the button tool bar and then close this window (or leave it open, doesn't hurt anything).



* The project should run satisfactorily – now you are ready to learn the material in this notes module.

**Menus**

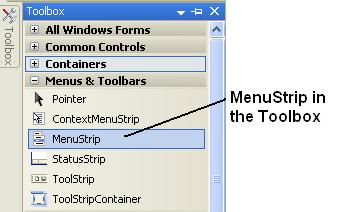
**Menus** are used in almost all application programs, often instead of buttons because they take less space on the form layout. They may also be used in addition to existing buttons.

Menus are built through use of a menu bar – these have menus that drop down to display menu items. **Menu items** have properties and methods and events, such as the **Name** and **Text** properties and the **Click** event.

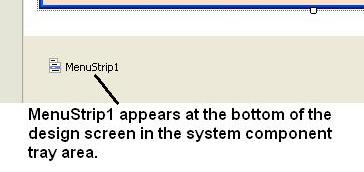
**MenuStrips** are container controls – you can add **ToolStripMenuItems**, **ToolStripComboBoxes**, **ToolStripSeparators**, and **ToolStripTextBoxes** – menu design is very flexible.

**Creating Menus**

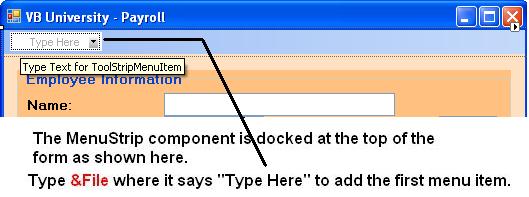
Begin by making room on the form for a **MenuStrip** control – highlight all existing controls on the form and drag them down to make room for a menu strip at the top of the form.



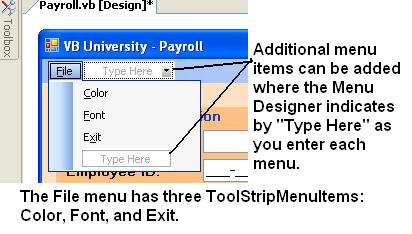
Add a **MenuStrip** **component** to the form – a control named **MenuStrip1** will display in the **system component tray** as shown in this figure.



When the **MenuStrip** control is selected, it displays as shown in this figure – the MenuStrip is docked (positioned) at the top of the form. This is called the Menu Designer – clicking anywhere on the form closes the Menu Designer – clicking the MenuStrip control again opens the Menu Designer.



Each time you type a menu entry such as **&File** where the **MenuStrip** says "**Type Here**", a new "**Type Here**" opens up both below and to the right of the current menu item as shown in this figure.



* The above figure shows three **ToolStripMenuItems** under the **FileToolStripMenuItem** – these are drop-down menus.
* Drop-down menus can be rearranged by dragging/dropping them until the menus are in the order desired. You can right-click a menu to delete it.
* Submenus – you can create submenus to any depth desired.

**Text Property**

As you type the words for a menu (called a **ToolStripMenuItem**), the typed value is saved to the menu item's **Text** property.

* The standard for ordering top-level menus is: **File**, **Edit**, and **View** (if used) followed by any specialty top-level menu items. The last two menus are always **Window** and **Help** if they are used.
* Menus should have **hot keys** – typing an ampersand (**&**) before a letter causes it to be underlined and to be a hot key for keyboard access by the **Alt+letter** method. Example: **&File**.

**Name Property**

VB automatically assigns useful names to the **ToolStripMenuItems**. Example:

* The **File** menu item will be named **FileToolStripMenuItem**.
* The **Exit** menu item will be named **ExitToolStripMenuItem**.

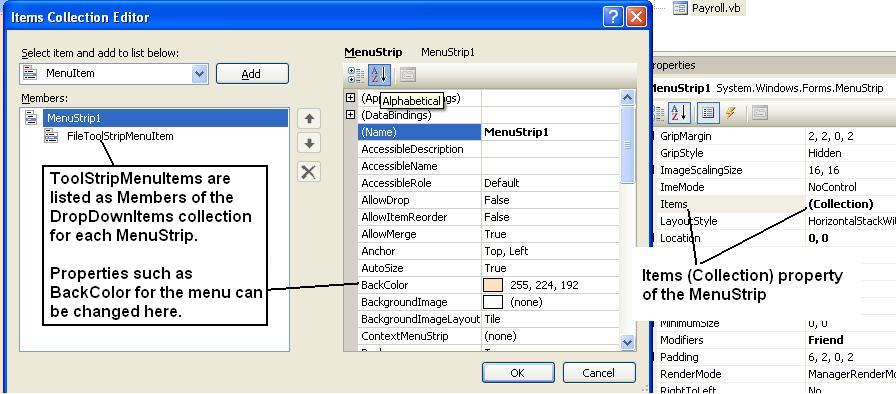
If you later change the **Text** property of a **ToolStripMenuItem**, VB does not change the name – if you want to rename the menu item, you must change the **Name** property.

**MenuStrip Items Collection**

Menu items (**ToolStripMenuItems**) added through the Menu Designer are added to an **Items** collection property of the **MenuStrip** control. There are **3 ways** to access the **Items Collection Editor** that is used to set menu properties:

1. Click the **MenuStrip** (not one of the **ToolStripMenuItems**) and Items collection property to use the Items Collection Editor.
2. Right-click the **MenuStrip** – select **Edit Items** from the context menu.
3. Click the **MenuStrip**'s smart-tag arrow shown at the end of the strip and select **Edit Items**.

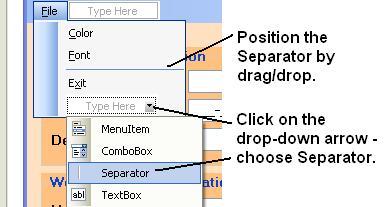
This figure shows the Items Collection Editor – individual ToolStripMenuItems are shown as **Members** of the **DropDownItems** collection in the left window pane – the right window pane shows properties of the selected menu item to be set.



**Separator Bars and Other Features**

A **Separator Bar** is a line drawn across a menu to separate items by purpose. Create them by:

* Access a "**Type Here**" and click the drop-down arrow shown in this figure.
* Select **Separator** from the drop-down list.



Notice in the figure above that you can also add the drop-down combo box and text box controls to a menu. You can also add a menu to other objects on a form, such as a group box control.

**In-Class Exercise**

Create the menus shown below for a new project.

* &File
  + &Color
  + &Font
  + Add a Separator bar here.
  + E&xit
* &Actions
  + &Compute
  + &Reset Form
  + &Total Salary Information
* &Help
  + &About

**Other Menu Properties & Information**

**Enabled:** All menus default the **Enabled** property to **True**. If a menu is disabled, it is grayed out at run time.

You can set the **Enabled** property at design time with the Property window or in code as shown here:

**TotalSalaryInformationToolStripMenuItem.Enabled = False**

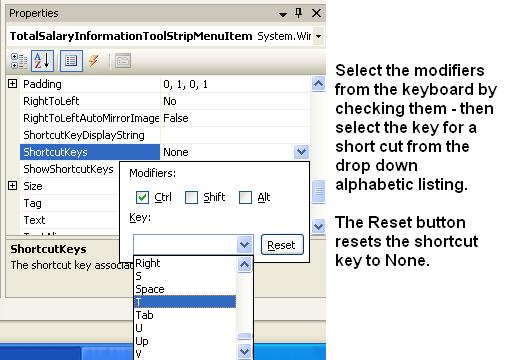
**Checked:** Set to **True** to cause a check mark to display beside a menu items – the item is **checked**. The **Checked** property defaults to **False**.

You can set the **Checked** property at design time or in code as shown here:

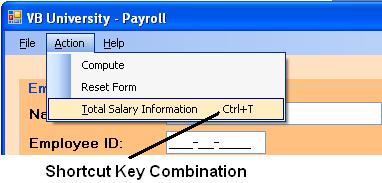
**TotalSalaryInformationToolStripMenuItem.Checked = True**

**ShortcutKeys and ShowShortcutKeys:** Shortcut keys are used to directly access a menu without going through the hot keys or without using the mouse to display the dropdown menu options.

Select the menu item, then select the **Properties** window – access the **ShortcutKeys** property and use the dropdown as shown in this figure to set a menu's shortcut key combination – here the key selected is **Ctrl+T** for the **TotalSalaryInformationToolStripMenuItem**. The **ShowShortcutKeys** property must be **True**.



The menu now looks like this figure.



**Coding a Menu Click Event**

The primary event coded for a menu is the **Click** event – code it as you would a button's click event sub procedure. Double-click the menu item to access the coding window.

This example code is for the **Help-About** menu – a message box is used to display information about the application.

**Private Sub AboutToolStripMenuItem\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles AboutToolStripMenuItem.Click**

**'Display about message box**

**Dim MessageString As String = "Programmed by Doug Bock" & ControlChars.NewLine & "Today's Date/Time: " & Date.Now.ToString**

**Dim TitleString As String = "About the Payroll Application"**

**'Display output message**

**MessageBox.Show(MessageString, TitleString , MessageBoxButtons.OK, MessageBoxIcon.Information)**

**End Sub**

* The message is stored to **MessageString**, a string variable by use of concatenation with the **ControlChars.NewLine** enumeration.
* The **Date.Now.ToString** method returns the current date and time for display.
* The title information is stored to the **TitleString** string variable.

**Calling an Event Sub Procedure from a Menu Click Event**

When a menu option and a button control are provided to perform the same process, such as the **Action menu**, **Compute option** and the **Compute Button**, you want these two options/controls to work exactly the same way. There are several options for coding the **Click** event for this menu.

1. Copy the code from the **Compute** button **Click** event sub procedure – an UNACCEPTABLE good option because you now have two identical pieces of code to be maintained if the code ever needs to be modified – this can lead to code that doesn't execute the same for each control (button and menu).
2. Call the **Compute** button **Click** event sub procedure – this is the best approach.

There are two satisfactory approaches for calling another Click event sub procedure.

1. Modify the **Handles** clause of the original sub procedure
2. Use the **PerformClick** method.

**Handles Clause:** This example shows how to code a **Compute** button's **Click** event sub procedure so that it handles two different events. This is the same technique used in an earlier module for multiple radio buttons.

* + The addition to the **Handles** clause is shown in red and highlighted with yellow.
  + This sub procedure will now execute for both of these Click events.

**Private Sub ComputeButton\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ComputeButton.Click, ComputeToolStripMenuItem.Click**

**PerformClick** **Method:** This example shows how to code a **Click** event for a control, such as a menu, so that it calls the **Click** event for another control, such as a button.

**Private Sub ExitToolStripMenuItem\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ExitToolStripMenuItem.Click**

**'Call the ResetForm button's Click event**

**ExitButton.PerformClick()**

**End Sub**

* **Caution:** The **PerformClick** method does not work well in conjunction with the **ErrorProvider** control. If you call an event such as the **Click** event for **ResetFormButton** and one of the controls such as a **TextBox** has an invalid value that has raised the **Validating** event for the **TextBox**, then instead of executing the **ResetFormButton\_Click** event, the **Validating** event will be triggered and the form will NOT reset. For this reason, the **Handles** clause is preferred.
* Which is better? Most programmers prefer modifying the **Handles** clause.

Complete the following actions:

* Code the Action menu's **Compute** and **Reset Form** **Exit** menu options – use the **Handles** clause modification for these two menu items.

**Private Sub ComputeButton\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ComputeButton.Click, ComputeToolStripMenuItem.Click**

**. . .**

**Private Sub ResetFormToolStripMenuItem\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ResetButton.Click, ResetFormToolStripMenuItem.Click**

**. . .**

* Code the Action menu's **Exit** menu option – use the **PerformClick** method.

**Private Sub ExitButton\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ExitButton.Click, ExitToolStripMenuItem.Click 'Call Exit Button with PerformClick**

**ExitButton.PerformClick()**

**End Sub**

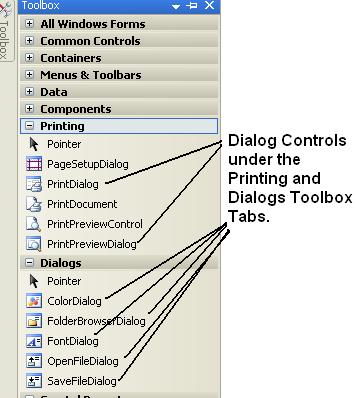
* Add the shortcut keys for several of the Action menu's **Compute** and **Reset Form** and the File menu’s **Exit** menu options – use keys of **Ctrl-C**, **Ctrl-R**, and **Ctrl-X** respectively.
* Test the program.

**Common Dialog Components**

Windows and Visual Basic offer predefined, standard dialog components for tasks such as:

* Changing colors and fonts
* Printing
* Opening and Saving files

The **Common Dialog Components** are found in both the **Dialogs** and **Printing** tabs of the Toolbox as shown in this figure:



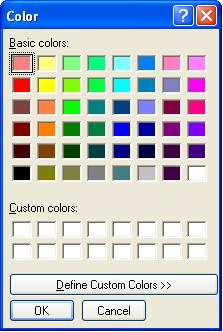
* Use the default VB names when coding with these component controls, such as **ColorDialog1** and **FontDialog1**.
* When using these controls, you only need one of each type of control for a form.
* Display all dialog components with the **ShowDialog** method, example:

**ColorDialog1.ShowDialog()**

**Color Dialog Component**

The **Color Dialog Component** is shown in this figure.

* The highlighted color (shown with light dotted lines around the color) is the current value of the **Color** property of the control.
* Selecting a color changes the value of the **Color** property of the control.



* **Cancel button** – cancels selection of a color – the **Color** property retains its original value.
* **Modal** – the Color Dialog is said to be modal – once it displays, you have to respond to it, you cannot switch to another window within the application – windows that allow you to switch between them are termed modeless.

This is the Click event sub procedure for the **ColorToolStripMenuItem**.

**Private Sub ColorToolStripMenuItem\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ColorToolStripMenuItem.Click**

**'Change the form's back color property.**

**'Stores current form color to the control's Color property.**

**'Display the Color Component, and save the new color**

**'to the form's BackColor property.**

**ColorDialog1.Color = Me.BackColor**

**ColorDialog1.ShowDialog()**

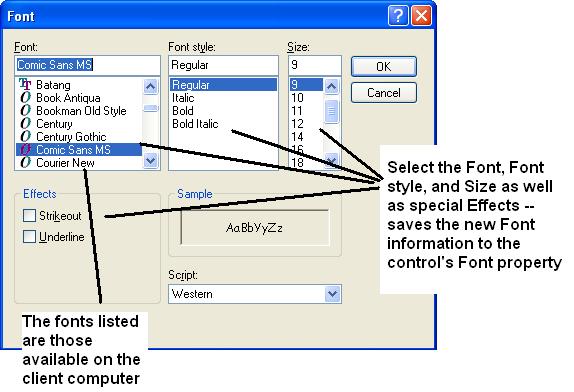
**Me.BackColor = ColorDialog1.Color**

**End Sub**

**Font Dialog Component**

This figure shows the **Font Dialog Component**.

* The highlighted font information reflects the current value of the **Font** property of the control.
* Selecting new font information changes the value of the **Font** property of the control.



After a selection is made, the **Font** property value can be assigned to the Font property of another control, such as the form itself as a control.

This is the Click event sub procedure for the **FontToolStripMenuItem**. This will change the font for the form and all controls on the form since they inherit their Font values from the form itself.

**Private Sub FontToolStripMenuItem\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles FontToolStripMenuItem.Click**

**'Change the font used for the entire form.**

**'Store current font to the control's Font property.**

**'Display the Font Component, and save the new font**

**'to the form's controls.**

**FontDialog1.Font = Me.Font**

**FontDialog1.ShowDialog()**

**Me.Font = FontDialog1.Font**

**End Sub**

**Possible errors:** If you set the font for individual controls such as GroupBox and TextBox controls, then the above code will not change the font for these controls. You will need to write additional code to set the font for individual controls such as the code highlighted here in yellow.

**FontDialog1.Font = Me.Font**

**FontDialog1.ShowDialog()**

**Me.Font = FontDialog1.Font**

**NameTextBox.Font = FontDialog1.Font**

**DepartmentTextBox.Font = FontDialog1.Font**

**MenuStrip1.Font = FontDialog1.Font**

**End Sub**

**In-Class Exercise Cont.**

Add both **Color Dialog** and **Font Dialog** components to the form – they will display in the system component tray under the form.

* Code the **File-Color** menu using the code shown above. The code should also change the **BackColor** of the **MenuStrip1** control.
* Code the **File-Font** menu using the code shown above. The code should also change the **Font** of the **MenuStrip1** control.

**Private Sub ColorToolStripMenuItem\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ColorToolStripMenuItem.Click**

**'Change the form's back color property.**

**'Stores current form color to the control's Color property.**

**'Display the Color Component, and save the new color**

**'to the form's BackColor property.**

**ColorDialog1.Color = Me.BackColor**

**ColorDialog1.ShowDialog()**

**Me.BackColor = ColorDialog1.Color**

**MenuStrip1.BackColor = ColorDialog1.Color**

**End Sub**

**Private Sub FontToolStripMenuItem\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles FontToolStripMenuItem.Click**

**'Change the font used for the entire form.**

**'Store current font to the control's Font property.**

**'Display the Font Component, and save the new font**

**'to the form's controls.**

**FontDialog1.Font = Me.Font**

**FontDialog1.ShowDialog()**

**Me.Font = FontDialog1.Font**

**MenuStrip1.Font = FontDialog1.Font**

**End Sub**

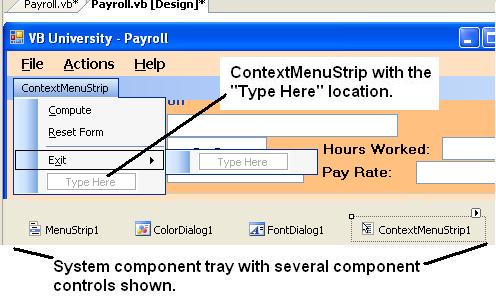
* Test the program. Note that when the font size changes, the form and all controls on the form automatically change to accommodate the new font size.

**Context Menus**

**Context menus** popup when you right-click an item – they are shortcut menus.

Add **ContextMenuStrip** **component** to the component tray.

* When the **ContextMenuStrip** **component** is selected, the form displays a place for you to type in menu items as shown in this figure.

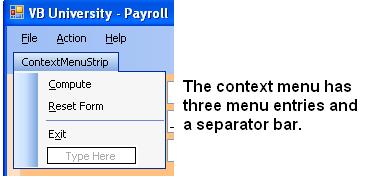


A form can have multiple context menus – so you can tailor the menu for different controls.

* The context menu for a form or control on a form is set by storing the name of the context menu to the **ContextMenuStrip property** for the control or form.
* If you have a single context menu – assign it to the form – this will cause most controls to have this as the assigned context menu.
* Some controls, such as textbox controls, have built-in context menus – they have the copy, cut, and paste text context menu.

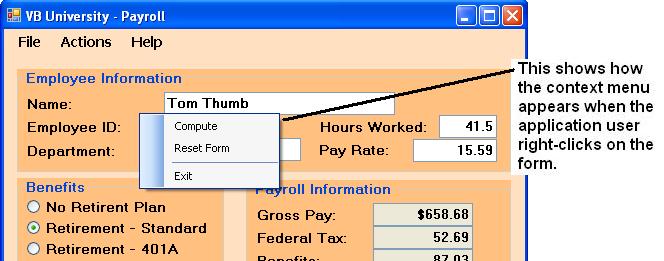
This context menu has been designed with three menu entries and a separator bar.

* Build the context menu by adding these three menu entries.

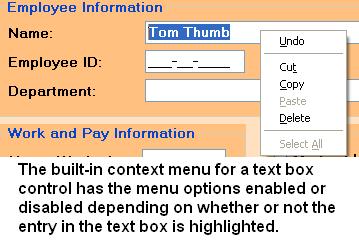


* Set the form's **ContextMenuStrip** property to **ContextMenuStrip1**.
* Run the project – confirm that the context menu displays whenever you right-click different parts of the form.
* Right-click a textbox control – observe that this control has a built-in context menu.

This figure shows right-clicking on the form (other than on a text box control).



This figure shows right-clicking on a text box control that has the entry highlighted.



Context menus are coded like other menus and buttons – by coding the **Click** event. You can also modify the **Handles** clause of an existing sub procedure to also handle the **Click** event of a context menu.

* In the **ExitButton\_Click** sub procedure shown below the **Handles** clause handles three different events.
* The **ExitToolStripMenuItem.Click** event for the main menu's **File-Exit** menu item is handled.
* The **ExiltToolStripMenuItem1.Click** event is the click event for the **context menu's Exit** menu item.

**Private Sub ExitButton\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ExitButton.Click, ExitToolStripMenuItem.Click, ExitToolStripMenuItem1.Click**

The **SourceControl** property of a context menu references the control to which the mouse is pointing at the time that the mouse is clicked.

* Used to enable changing a specific control's property settings.
* This example changes the **ForeColor** property of a control to which the mouse points.

**Private Sub ColorToolStripMenuItem1\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ColorToolStripMenuItem1.Click**

**'Change color of object**

**'Send color to dialog box**

**ColorDialog1.Color = ContextMenuStrip1.SourceControl.ForeColor**

**ColorDialog1.ShowDialog()**

**'Display new color**

**ContextMenuStrip1.SourceControl.ForeColor = ColorDialog1.Color**

**End Sub**

If you have not done so already, add a **ContextMenuStrip** component to the form – it will display in the system component tray and is automatically named **ContextMenuStrip1**.

* Add four context menu items: **Compute**, **Reset** **Form**, **Color**, and **Exit**.
* Add two separator bars, one between **ResetForm** and **Color** and one between **Color** and **Exit**.
* Set the **ContextMenuStrip** property of the form to **ContextMenuStrip1**.
* Set the **ContextMenuStrip** property of the **NameTextBox** text box control to **ContextMenuStrip1**.
* Modify the **Handles** clauses for the existing sub procedures for the corresponding **Compute**, **Reset** **Form**, and **Exit** Button controls so that the sub procedures also handle the context menu items.
* Also, modify the **Handles** clauses for the existing sub procedures to handle the menu items under the **File** and **Action** **ToolStripMenuItems** that correspond to the existing Button controls.
* Add code for the **Click** event for the **Color** context menu item as shown above.
* Add a **Color** menu item to the context menu and add the code shown above to the program.

Test the program. Ensure that each menu functions properly.

* Enter data and click the context menu to compute, and then reset the form.
* Type a name into the **NameTextBox** control, then right-click the textbox and select the **Color** menu item – when the color dialog window displays, change the color and click OK. Observe that the color of the name's text changes.
* Use the context menu to exit the program.

**Possible Errors:** Did you notice that if you do not tab away from the **NameTextBox**  control (or any other data entry control) when you reset the form, the **Validating** event for the control is triggered and the **ErrorProvider** control displays an error and locks the focus into the **NameTextBox** ? We will fix this problem in the next section.

**General Procedures**

Sometimes a programming module will become quite large, and it makes sense from a logical design perspective to divide the module into two or more separate procedures.

* You can store the code to a separate procedure, and then call the code from multiple button and menu control click events.
* You can also develop a procedure that you "reuse" by calling it from either a context menu or regular menu or button that is clicked.
* This is a third way to avoid duplicating (copying) code in a program, along with the **PerformClick** method and the alteration of the **Handles** clause.

There are two types of **general procedures**:

* **Sub procedures** that perform actions, but are not associated with any specific event such as a click event. Use sub procedures to set property values for a series of objects.
* **Function procedures** that perform actions, are not associated with any specific event, and that also return a value. Use functions to calculate values that are returned to the calling procedure.

Both sub procedures and functions are treated as **methods** (members) of the form and are listed with the namespace for the form.

Neither sub procedures or function procedures are associated with an event – they are separate procedures.

**Creating a Sub Procedure**

Type the following line and press the Enter key (give the sub procedure some unique, meaningful name):

**Private Sub *ProcedureName***

* This causes VB to add parentheses and the **End Sub** statement. You can then add your code inside the procedure.
* In addition to **Private**, there are **Public**, **Friend**, and **Protected** procedures that are covered in later notes.

**Example Sub Procedure – SetValidationStatus**

The **SetValidationStatus** sub procedure is an example sub procedure with a parameter.

* It takes an incoming parameter named **ValueBoolean**  that can be either **True** or **False**.
* When **ValueBoolean**  is **False**, this turns off the **CausesValidation** property of the data input controls, and when **True**, it turns on the **CausesValidation** property.
* Because it is possible that an **ErrorProvider** icon may be displayed, this sub procedure also clears all possible **ErrorProvider** icon messages.

**Private Sub SetValidationStatus(ByVal ValueBoolean As Boolean)**

**'Set validation status according to ValueBoolean**

**'to turn control validation on/off**

**NameTextBox.CausesValidation = ValueBoolean**

**EmployeeIDMaskedTextBox.CausesValidation = ValueBoolean**

**DepartmentTextBox.CausesValidation = ValueBoolean**

**HoursTextBox.CausesValidation = ValueBoolean**

**PayRateTextBox.CausesValidation = ValueBoolean**

**'Clear all possible error provider icon messages**

**ErrorProvider1.SetError(NameTextBox, "")**

**ErrorProvider1.SetError(EmployeeIDMaskedTextBox, "")**

**ErrorProvider1.SetError(DepartmentTextBox, "")**

**ErrorProvider1.SetError(HoursTextBox, "")**

**ErrorProvider1.SetError(PayRateTextBox, "")**

**End Sub**

* The **ValueBoolean**  variable is termed a **parameter**. It is declared as type **Boolean** in the parameter list (inside parentheses) of the sub procedure.

How is the **SetValidationStatus** sub procedure used? You can call it from the **ResetButton\_Click** event sub procedure – this Click event is modified to call (use) the **SetValidationStatus** sub procedure. The new code is shown below highlighted in yellow.

* The first time the sub procedure is called, it is sent the value **False** – this will disable validating the data entry controls.
* The second time the sub procedure is called, it is sent the value **True** – this again enables validating the data entry controls.
* Turning validation off then back on enables resetting the form without triggering validation of the controls.
* Using a separate sub procedure eliminates the need to duplicate the code twice within the **ResetButton\_Click** event sub procedure.

**Private Sub ResetButton\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ResetButton.Click, ResetFormToolStripMenuItem1.Click**

**. . . <the upper ½ of the sub procedure is not shown here**

**'Uncheck benefits checkboxes**

**MedicalCheckBox.Checked = False**

**LifeCheckBox.Checked = False**

**DentalCheckBox.Checked = False**

**'Call SetValidationStatus procedure to disable**

**'validating data entry controls**

**SetValidationStatus(False)**

**'Set focus to name textbox**

**NameTextBox.Focus()**

**'Call SetValidationStatus procedure to enable**

**'validating data entry controls**

**SetValidationStatus(True)**

**End Sub**

**Calling SetValidationStatus during Exit**

One problem with the **ErrorProvider** control is that it can be difficult to NOT cause validation. This example illustrates calling the SetValidationStatus during the **Exit** Button control’s Click event.

* The **SetValidationStatus** sub procedure is called with an argument of **False** to turn off validation for the form so the form will exit if the application user responds Yes to the message box.
* **SetValidationStatus** is called with an argument of **True** to ensure validation is on for the form otherwise.

**Private Sub ExitButton\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ExitButton.Click, ExitToolStripMenuItem.Click, ExitToolStripMenuItem1.Click**

**'Close the form if the system user responds Yes**

**Dim MessageString As String = "Do you want to close the form?"**

**Dim ButtonDialogResult As DialogResult = MessageBox.Show(MessageString, "Quit?", MessageBoxButtons.YesNo, MessageBoxIcon.Question, MessageBoxDefaultButton.Button2)**

**If ButtonDialogResult = Windows.Forms.DialogResult.Yes Then**

**'Turn off validation to exit**

**SetValidationStatus(False)**

**Me.Close()**

**Else**

**'Ensure validation is on**

**SetValidationStatus(True)**

**End If**

**End Sub**

**Rules About Passing Arguments (Parameters)**

* Arguments passed to a sub procedure must be the **same type of data** as the sub procedure receives, i.e., you wouldn't pass a football player a beach ball – you'd pass a football!.
* Argument values **must be passed** if the sub procedure has an argument list.
* A sub procedure can have as many arguments as are needed, and the sequence and data type must match between the calling and called procedures.

**ByVal Versus ByRef**

Arguments can be passed **ByVal** or **ByRef**.

* **ByVal** sends a copy of the argument's value to the procedure so the procedure does not alter the original copy of the value.
* **ByRef** sends the memory location of the value to the procedure so that if the value is modified, the original copy is modified.
* **ByVal** is the **default** if it is not specified.

Examples:

**Private Sub SetValidationStatus(ByVal ValueBoolean As Boolean)**

**Private Sub SetValidationStatus(ByRef ValueBoolean As Boolean)**

**In-Class Exercise Cont. – Sub Procedure Coding**

* Add a sub procedure named **SetValidationStatus** to your project.
* Modify the existing procedures **ResetButton\_Click** event sub procedure to call the **SetValidationStatus** sub procedure.

**Creating a Function**

Functions you create are called **User-Defined Functions**.

* You have already used some functions such as the **IsNumeric** function.
* Functions are used to simplify the coding of a program – you code them to perform some task such as validating data or computing sales tax or computing a salesperson's commission amount
* **Functions always return a value**. For example, the **IsNumeric** returns a **Boolean** value of **True** or **False**.

Create a function by typing:

**Private Function *FunctionName*() As Datatype**

You must specify the **Datatype** that the function returns – this is done with the **As Datatype** clause in the function declaration as highlighted above.

This function computes the sales commission amount.

* Functions ALWAYS return a value – you may optionally may pass values to a function – these are called parameters
* In the example below the function named **Commission** receives a **decimal parameter** that is passed to the function – this is the amount of the sale as a decimal value – the value incoming to the function is stored in the variable named **SalesAmountDecimal**.

**Private Function Commission(ByVal SalesAmountDecimal As Decimal) As Decimal**

**'Declare internal variables/constants**

**Const SaleLevelLowDecimal As Decimal = 1000D**

**Const SaleLevelMediumDecimal As Decimal = 2500D**

**'Calculate commission based on amount of sale**

**If SalesAmountDecimal <= SaleLevelLowDecimal Then**

**Commission = 0.05D \* SalesAmountDecimal**

**ElseIf SalesAmountDecimal <= SaleLevelMediumDecimal Then**

**Commission = 0.1D \* SalesAmountDecimal**

**Else**

**Commission = 0.2D \* SalesAmountDecimal**

**End If**

**End Function**

There are two ways to return a value from a function:

1. Set the function name to the value to be returned inside the function as was done above – here the function name **Commission** is set to the value to be returned.
2. Use the optional **Return** statement as is done below.

**Private Function Commission(ByVal SalesAmountDecimal As Decimal) As Decimal**

**'Declare internal variables/constants**

**Const SaleLevelLowDecimal As Decimal = 1000D**

**Const SaleLevelMediumDecimal As Decimal = 2500D**

**Dim CommissionDecimal As Decimal**

**'Calculate commission based on amount of sale**

**If SalesAmountDecimal <= SaleLevelLowDecimal Then**

**CommissionDecimal = 0.05D \* SalesAmountDecimal**

**ElseIf SalesAmountDecimal <= SaleLevelMediumDecimal Then**

**CommissionDecimal = 0.1D \* SalesAmountDecimal**

**Else**

**CommissionDecimal = 0.2D \* SalesAmountDecimal**

**End If**

**Return CommissionDecimal**

**End Function**

**Calling a Function**

This procedure shows how the function named **Commission** is called from a "calling" procedure.

* A function that returns a value must be used in either: (1) an If statement or, (2) on the right-hand side of an assignment statement.
* The value returned is stored to the variable in the assignment statement (here the value is returned and stored to **CommissionAmountDecimal**).
* The parameter(s) must be passed to the function by the calling statement.

**Private Sub ComputeButton\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ComputeButton.Click, ComputeToolStripMenuItem.Click, ComputeToolStripMenuItem1.Click**

**Dim CommissionAmountDecimal, SalesDecimal As Decimal**

**If IsNumeric(SalesTextBox.Text) Then**

**'Convert sales text box amount**

**SalesDecimal = Decimal.Parse(SalesTextBox.Text, Globalization.NumberStyles.Currency)**

**'Here is the Call to the Commission function**

**CommissionAmountDecimal = Commission(SalesDecimal)**

**commissionTextBox.Text = CommissionAmountDecimal.ToString("C")**

**End If**

**. . . more code follows**

Just like sub procedures, a function can have zero, one, or more arguments in the parameter list. The calling and called procedure must pass/receive the same type of data for each argument in the parameter list.

**ComputeGrossPay Function**

The **ComputeGrossPay** function shown here simplifies the Compute button's click event by moving the computations for gross pay to a separate function.

* This is an example of a function that has **no parameter values** passed to it.
* When working on a project team, the project leader could assign you the task of writing the click event code for the **Compute** button, and another programmer could be assigned the task of writing the **ComputeGrossPay** function – this divides up the work load for a large programming project.
* The function returns a decimal value by storing the computed value of gross pay to the name of the function (highlighted in yellow).
* You must remove the declaration of **HoursDecimal**  and **PayRateDecimal** from the Compute button's Click event as these two variables are now only used inside the function.

**Private Function ComputeGrossPay() As Decimal**

**'Parse textbox values to memory variables**

**Dim HoursDecimal As Decimal = Decimal.Parse(HoursTextBox.Text, Globalization.NumberStyles.Number)**

**Dim PayRateDecimal As Decimal = Decimal.Parse(PayRateTextBox.Text, Globalization.NumberStyles.Currency)**

**'Gross pay is computed and returned by assigning the value**

**'to the name of the function**

**If HoursDecimal <= 40D Then 'pay only regular time**

**ComputeGrossPay = Decimal.Round(HoursDecimal \* PayRateDecimal, 2)**

**Else 'pay regular + overtime**

**ComputeGrossPay = Decimal.Round((40D \* PayRateDecimal) + ((HoursDecimal - 40D) \* PayRateDecimal \* 1.5D), 2)**

**End If**

**End Function**

The **ComputeGrossPay** function is called as shown below in this coding segment from the Compute button's Click event

* Note the use of the function name on the right-side of the assignment statement.
* Control will transfer to the function, the value of gross pay will be computed, and then the value returned by the function is stored to the variable **GrossPayDecimal** .

**'Call function ComputeGrossPay**

**GrossPayDecimal = ComputeGrossPay()**

**. . . more code follows**

**ComputeFederalTax Function**

The **ComputeFederalTax** function is an example of a function with a **single** **input parameter**.

* This is another way to simplify the Compute button's click event.
* The input parameter **GrossPayDecimal**  is passed to the **ComputeFederalTax** function.
* Internal variables and constants used within the function are declared. The module-level constants related to computing federal tax are moved into the function and deleted from the general declarations area.
* The same **Select Case** structure used earlier in the **Compute** button's click event is cut/pasted here and modified as necessary to properly compute the amount of federal tax due.
* The value of **FederalTaxDecimal**  is returned to the calling procedure with a **Return** statement (highlighted in red).

**Private Function ComputeFederalTax(ByVal GrossPayDecimal As Decimal) As Decimal**

**'Tax rate constants**

**Const TaxRate08Decimal As Decimal = 0.08D**

**Const TaxRate18Decimal As Decimal = 0.18D**

**Const TaxRate28Decimal As Decimal = 0.28D**

**Const TaxLevel08Decimal As Decimal = 985D**

**Const TaxLevel18Decimal As Decimal = 2450D**

**'Declare variables used within the function**

**Dim FederalTaxDecimal As Decimal**

**'Compute amount of tax based on the amount of pay**

**Select Case GrossPayDecimal**

**Case Is <= TaxLevel08Decimal '8% tax bracket**

**FederalTaxDecimal = Decimal.Round(TaxRate08Decimal \* GrossPayDecimal, 2)**

**Case Is <= TaxLevel18Decimal '18% tax bracket**

**FederalTaxDecimal = Decimal.Round(TaxRate18Decimal \* GrossPayDecimal, 2)**

**Case Else '28% tax bracket**

**FederalTaxDecimal = Decimal.Round(taxRate28Decimal \* GrossPayDecimal, 2)**

**End Select**

**'Return a value**

**Return FederalTaxDecimal**

**End Function**

The Compute button's click event is modified to call the function by passing the value of the variable **GrossPayDecimal**  (highlighted in yellow) to the function and storing the returned value to the variable **FederalTaxDecimal**  (highlighted in red).

* While the variable **GrossPayDecimal**  passed from the Compute button’s click event is the same name as the input parameter in the **ComputeFederalTax** function, the two names do **NOT** have to be the same – just the data type has to be the same.
* This is also true for the variable named **FederalTaxDecimal**  here and in the function – the two variables are actually different variables!

**. . .**

**'Compute the federal tax**

**FederalTaxDecimal = ComputeFederalTax(GrossPayDecimal )**

**ComputeBenefits Function**

The **ComputeBenefits** function is another example of a function with an **input parameter**.

* This is another way to simplify the Compute button's click event.
* This function accepts an input parameter **GrossPayDecimal**  that is passed to the function from the calling procedure.
* One internal variable used within the function is declared.
* The same **If** structure used earlier in the **Compute** button's click event is cut/pasted here and modified as necessary to properly compute the amount of the benefit deduction.
* The value of **BenefitsCostDecimal**  is returned to the calling procedure with a **Return** statement (highlighted in red).

**Private Function ComputeBenefitsDeduction(ByVal GrossPayDecimal As Decimal) As Decimal**

**'Benefit constants**

**Const MedicalRateDecimal As Decimal = 35.75D**

**Const LifeRateDecimal As Decimal = 18.35D**

**Const DentalRateDecimal As Decimal = 4D**

**'Declare variables used within the function**

**Dim BenefitsCostDecimal As Decimal**

**'Compute insurance benefits deduction**

**If MedicalCheckBox.Checked Then**

**BenefitsCostDecimal += MedicalRateDecimal 'selected medical insurance**

**End If**

**If LifeCheckBox.Checked Then**

**BenefitsCostDecimal += LifeRateDecimal 'selected life insurance**

**End If**

**If DentalCheckBox.Checked Then**

**BenefitsCostDecimal += DentalRateDecimal 'selected dental insurance**

**End If**

**'Use the retirement rate set in the CheckedChanged event**

**'for the retirement radio button controls**

**BenefitsCostDecimal += Decimal.Round(GrossPayDecimal \* RetirementRateDecimal, 2)**

**'Return cost of benefits**

**Return BenefitsCostDecimal**

**End Function**

The **Compute** button's **Click** event is modified to call the function by passing the value of the variable **GrossPayDecimal**  (highlighted in red) to the function and storing the returned value to the variable **BenefitsCostDecimal**  (highlighted in red).

**'Compute the benefits deduction.**

**BenefitsCostDecimal = ComputeBenefitsDeduction(GrossPayDecimal )**

* Note that the variable **BenefitsCostDecimal**  inside the function is a different variable than the **BenefitsCostDecimal**  variable inside the **Compute** button's click event – they happen to have the same name, but are, in fact different variables.

The complete **Compute** button **Click** event sub procedure is shown here so that you can see how all of the function calls have simplified the sub procedure.

**Private Sub ComputeButton\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ComputeButton.Click, ComputeToolStripMenuItem.Click, ComputeToolStripMenuItem1.Click**

**Try**

**'Declare variables and constants**

**Dim GrossPayDecimal, FederalTaxDecimal, BenefitsCostDecimal, NetPayDecimal As Decimal**

**'Call function ComputeGrossPay - no need to pass**

**'any parameter values to the function**

**GrossPayDecimal = ComputeGrossPay()**

**'Compute the federal tax - pass the value of**

**'GrossPayDecimal as a parameter to the function**

**FederalTaxDecimal = ComputeFederalTax(GrossPayDecimal)**

**'Compute the benefits deduction - pass the value of**

**'GrossPayDecimal as a parameter to the function**

**BenefitsCostDecimal = ComputeBenefitsDeduction(GrossPayDecimal)**

**'Compute the net pay – no need to round because**

**'all values are already rounded**

**NetPayDecimal = GrossPayDecimal - FederalTaxDecimal - BenefitsCostDecimal**

**'Display output – this shows all four outputed values**

**GrossPayTextBox.Text = GrossPayDecimal.ToString("C2")**

**FederalTaxTextBox.Text = FederalTaxDecimal.ToString("N2")**

**BenefitsTextBox.Text = BenefitsCostDecimal.ToString("N2")**

**NetPayTextBox.Text = NetPayDecimal.ToString("C2")**

**Catch ex As Exception**

**MessageBox.Show("Check to ensure no data values are missing." & ControlChars.NewLine & ex.Message, "Compute Button Error", MessageBoxButtons.OK, MessageBoxIcon.Error)**

**End Try**

**End Sub**

**In-Class Exercise**

Modify the **Payroll** project as follows:

* Add the **ComputeGrossPay**, **ComputeFederalTax**, and the **ComputeBenefits** functions shown above.
* Modify the **Compute** button's click event to call the functions and store the values returned.

Test your project for various gross pay and benefit levels to confirm that it still produces exactly the same output as before.

**Using Regions to Organize Code**

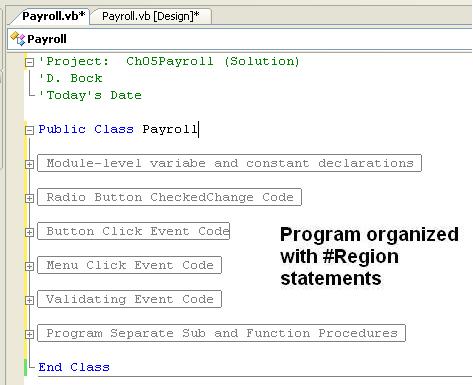
The **#Region** statement can be used to organize the code in a program as the size of the program grows.

* This **#Region** statement is used to create a region for the storage of sub and function procedures.
* The sub and function procedures can be cut/pasted into the region and the region can be collapsed.

**#Region " Program Separate Sub and Function Procedures "**

**#End Region**

This figure shows the main program organized with regions.



**Solution to In-Class Exercise**

**'Project: Ch05Payroll (Solution)**

**'D. Bock**

**'Today's Date**

**Public Class Payroll**

**#Region " Module-level variable and constant declarations "**

**'Module level variable/constant declarations**

**Private RetirementRateDecimal As Decimal**

**#End Region**

**#Region " Radio Button CheckedChange Code "**

**Private Sub NoneRadioButton\_CheckedChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles NoneRadioButton.CheckedChanged, Retirement401ARadioButton.CheckedChanged, RetirementStandardRadioButton.CheckedChanged**

**'Declare retirement benefit constants**

**Const RetirementStandardDecimal As Decimal = 0.05D**

**Const Retirement401ADecimal As Decimal = 0.08D**

**'Create a radio button in memory and store the values of sender to it**

**Dim CheckedRadioButton As RadioButton = CType(sender, RadioButton)**

**'Use Select Case to evaluate the name of the radio button**

**'to decide which controls to enable/disable**

**Select Case CheckedRadioButton.Name**

**Case "NoneRadioButton" 'Cost is zero**

**RetirementRateDecimal = 0D**

**Case "RetirementStandardRadioButton" 'Standard rate**

**RetirementRateDecimal = RetirementStandardDecimal**

**Case "Retirement401ARadioButton" '401A rate**

**RetirementRateDecimal = Retirement401ADecimal**

**End Select**

**End Sub**

**#End Region**

**#Region " Button Click Event Code"**

**Private Sub ComputeButton\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ComputeButton.Click, ComputeToolStripMenuItem.Click, ComputeToolStripMenuItem1.Click**

**Try**

**'Declare variables and constants**

**Dim GrossPayDecimal, FederalTaxDecimal, BenefitsCostDecimal, NetPayDecimal As Decimal**

**'Call function ComputeGrossPay - no need to pass**

**'any parameter values to the function**

**GrossPayDecimal = ComputeGrossPay()**

**'Compute the federal tax - pass the value of**

**'GrossPayDecimal as a parameter to the function**

**FederalTaxDecimal = ComputeFederalTax(GrossPayDecimal)**

**'Compute the benefits deduction - pass the value of**

**'GrossPayDecimal as a parameter to the function**

**BenefitsCostDecimal = ComputeBenefitsDeduction(GrossPayDecimal)**

**'Compute the net pay – no need to round because**

**'all values are already rounded**

**NetPayDecimal = GrossPayDecimal - FederalTaxDecimal - BenefitsCostDecimal**

**'Display output – this shows all four outputed values**

**GrossPayTextBox.Text = GrossPayDecimal.ToString("C2")**

**FederalTaxTextBox.Text = FederalTaxDecimal.ToString("N2")**

**BenefitsTextBox.Text = BenefitsCostDecimal.ToString("N2")**

**NetPayTextBox.Text = NetPayDecimal.ToString("C2")**

**Catch ex As Exception**

**MessageBox.Show("Check to ensure no data values are missing." & ControlChars.NewLine & ex.Message, "Compute Button Error", MessageBoxButtons.OK, MessageBoxIcon.Error)**

**End Try**

**End Sub**

**Private Sub ExitButton\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ExitButton.Click, ExitToolStripMenuItem.Click, ExitToolStripMenuItem1.Click**

**'Close the form if the system user responds Yes**

**Dim MessageString As String = "Do you want to close the form?"**

**Dim ButtonDialogResult As DialogResult = MessageBox.Show(MessageString, "Quit?", MessageBoxButtons.YesNo, MessageBoxIcon.Question, MessageBoxDefaultButton.Button2)**

**If ButtonDialogResult = Windows.Forms.DialogResult.Yes Then**

**'Turn off validation to exit**

**SetValidationStatus(False)**

**Me.Close()**

**Else**

**'Ensure validation is on**

**SetValidationStatus(True)**

**End If**

**End Sub**

**Private Sub ResetButton\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ResetButton.Click, ResetFormToolStripMenuItem1.Click**

**'Clear all textbox controls**

**NameTextBox.Clear()**

**EmployeeIDMaskedTextBox.Clear()**

**DepartmentTextBox.Clear()**

**HoursTextBox.Clear()**

**PayRateTextBox.Clear()**

**GrossPayTextBox.Clear()**

**FederalTaxTextBox.Clear()**

**BenefitsTextBox.Clear()**

**NetPayTextBox.Clear()**

**'Reset retirement benefits status to none**

**NoneRadioButton.Checked = True**

**'Uncheck benefits checkboxes**

**MedicalCheckBox.Checked = False**

**LifeCheckBox.Checked = False**

**DentalCheckBox.Checked = False**

**'Call SetValidationStatus procedure to disable**

**'validating data entry controls**

**SetValidationStatus(False)**

**'Set focus to name textbox**

**NameTextBox.Focus()**

**'Call SetValidationStatus procedure to enable**

**'validating data entry controls**

**SetValidationStatus(True)**

**End Sub**

**#End Region**

**#Region " Menu Click Event Code "**

**Private Sub AboutToolStripMenuItem\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles AboutToolStripMenuItem.Click**

**'Display about message box**

**Dim MessageString As String = "Programmed by Doug Bock" & ControlChars.NewLine & "Today's Date/Time: " & Date.Now.ToString**

**Dim TitleString As String = "About the Payroll Application"**

**'Display output message**

**MessageBox.Show(MessageString, TitleString, MessageBoxButtons.OK, MessageBoxIcon.Information)**

**End Sub**

**Private Sub ResetFormToolStripMenuItem\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ResetFormToolStripMenuItem.Click**

**'Call Reset Button with PerformClick**

**ResetButton.PerformClick()**

**End Sub**

**Private Sub ColorToolStripMenuItem\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ColorToolStripMenuItem.Click**

**'Change the form's back color property.**

**'Stores current form color to the control's Color property.**

**'Display the Color Component, and save the new color**

**'to the form's BackColor property.**

**ColorDialog1.Color = BackColor**

**ColorDialog1.ShowDialog()**

**BackColor = ColorDialog1.Color**

**MenuStrip1.BackColor = ColorDialog1.Color**

**End Sub**

**Private Sub FontToolStripMenuItem\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles FontToolStripMenuItem.Click**

**'Change the font used for the entire form.**

**'Store current font to the control's Font property.**

**'Display the Font Component, and save the new font**

**'to the form's controls.**

**FontDialog1.Font = Font**

**FontDialog1.ShowDialog()**

**Font = FontDialog1.Font**

**MenuStrip1.Font = FontDialog1.Font**

**End Sub**

**Private Sub ColorToolStripMenuItem1\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ColorToolStripMenuItem1.Click**

**'Change color of object**

**'Send color to dialog box**

**ColorDialog1.Color = ContextMenuStrip1.SourceControl.ForeColor**

**ColorDialog1.ShowDialog()**

**'Display new color**

**ContextMenuStrip1.SourceControl.ForeColor = ColorDialog1.Color**

**End Sub**

**#End Region**

**#Region " Validating Event Code "**

**Private Sub NameTextBox\_Validating(ByVal sender As Object, ByVal e As System.ComponentModel.CancelEventArgs) Handles NameTextBox.Validating**

**'Validate the employee information groupbox**

**If NameTextBox.Text = String.Empty Then**

**'Required employee name is missing**

**'Cancel the event, show the error message.**

**e.Cancel = True**

**ErrorProvider1.SetError(NameTextBox, "Name is required")**

**Else**

**'Clear error provider message**

**ErrorProvider1.SetError(NameTextBox, "")**

**End If**

**End Sub**

**Private Sub EmployeeIDMaskedTextBox\_Validating(ByVal sender As Object, ByVal e As System.ComponentModel.CancelEventArgs) Handles EmployeeIDMaskedTextBox.Validating**

**'Validate the employee information groupbox**

**If EmployeeIDMaskedTextBox.Text = " - -" Then**

**'Required employee ID is missing**

**e.Cancel = True**

**ErrorProvider1.SetError(EmployeeIDMaskedTextBox, "Employee ID is required")**

**Else**

**'Clear error provider message**

**ErrorProvider1.SetError(EmployeeIDMaskedTextBox, "")**

**End If**

**End Sub**

**Private Sub DepartmentTextBox\_Validating(ByVal sender As Object, ByVal e As System.ComponentModel.CancelEventArgs) Handles DepartmentTextBox.Validating**

**If DepartmentTextBox.Text = String.Empty Then**

**'Required department is missing**

**e.Cancel = True**

**ErrorProvider1.SetError(DepartmentTextBox, "Department is required")**

**Else**

**'Clear error provider message**

**ErrorProvider1.SetError(DepartmentTextBox, "")**

**End If**

**End Sub**

**Private Sub HoursTextBox\_Validating(ByVal sender As Object, ByVal e As System.ComponentModel.CancelEventArgs) Handles HoursTextBox.Validating**

**'Declare constant used in this sub procedure**

**Const MaxHoursDecimal As Decimal = 60D**

**'Validate TextBox value**

**If IsNumeric(HoursTextBox.Text) = False OrElse (Decimal.Parse(HoursTextBox.Text, Globalization.NumberStyles.Number) <= 0D Or Decimal.Parse(HoursTextBox.Text, Globalization.NumberStyles.Number) > MaxHoursDecimal) Then**

**'Hours must be numeric and within allowable range**

**e.Cancel = True**

**ErrorProvider1.SetError(HoursTextBox, "Hours worked must be a number between 0 and " & MaxHoursDecimal.ToString)**

**HoursTextBox.SelectAll()**

**Else**

**'Clear error provider message**

**ErrorProvider1.SetError(HoursTextBox, "")**

**End If**

**End Sub**

**Private Sub PayRateTextBox\_Validating(ByVal sender As Object, ByVal e As System.ComponentModel.CancelEventArgs) Handles PayRateTextBox.Validating**

**If IsNumeric(PayRateTextBox.Text) = False OrElse Decimal.Parse(PayRateTextBox.Text, Globalization.NumberStyles.Currency) <= 0D Then**

**'Pay rate must be numeric and greater than zero**

**e.Cancel = True**

**ErrorProvider1.SetError(PayRateTextBox, "Pay rate must be a number and greater than zero")**

**PayRateTextBox.SelectAll()**

**Else**

**'Clear error provider message**

**ErrorProvider1.SetError(PayRateTextBox, "")**

**End If**

**End Sub**

**Private Sub Payroll\_FormClosing(ByVal sender As Object, ByVal e As System.Windows.Forms.FormClosingEventArgs) Handles Me.FormClosing**

**'This enables closing a form that contains invalid data**

**e.Cancel = False**

**End Sub**

**#End Region**

**#Region " Program Separate Sub and Function Procedures "**

**Private Sub SetValidationStatus(ByVal ValueBoolean As Boolean)**

**'Set validation status according to ValueBoolean**

**'to turn control validation on/off**

**NameTextBox.CausesValidation = ValueBoolean**

**EmployeeIDMaskedTextBox.CausesValidation = ValueBoolean**

**DepartmentTextBox.CausesValidation = ValueBoolean**

**HoursTextBox.CausesValidation = ValueBoolean**

**PayRateTextBox.CausesValidation = ValueBoolean**

**'Clear all possible error provider icon messages**

**ErrorProvider1.SetError(NameTextBox, "")**

**ErrorProvider1.SetError(EmployeeIDMaskedTextBox, "")**

**ErrorProvider1.SetError(DepartmentTextBox, "")**

**ErrorProvider1.SetError(HoursTextBox, "")**

**ErrorProvider1.SetError(PayRateTextBox, "")**

**End Sub**

**Private Function ComputeGrossPay() As Decimal**

**'Parse textbox values to memory variables**

**Dim HoursDecimal As Decimal = Decimal.Parse(HoursTextBox.Text, Globalization.NumberStyles.Number)**

**Dim PayRateDecimal As Decimal = Decimal.Parse(PayRateTextBox.Text, Globalization.NumberStyles.Currency)**

**'Gross pay is computed and returned by assigning the value**

**'to the name of the function**

**If HoursDecimal <= 40D Then 'pay only regular time**

**ComputeGrossPay = Decimal.Round(HoursDecimal \* PayRateDecimal, 2)**

**Else 'pay regular + overtime**

**ComputeGrossPay = Decimal.Round((40D \* PayRateDecimal) + ((HoursDecimal - 40D) \* PayRateDecimal \* 1.5D), 2)**

**End If**

**End Function**

**Private Function ComputeFederalTax(ByVal GrossPayDecimal As Decimal) As Decimal**

**'Tax rate constants**

**Const TaxRate08Decimal As Decimal = 0.08D**

**Const TaxRate18Decimal As Decimal = 0.18D**

**Const TaxRate28Decimal As Decimal = 0.28D**

**Const TaxLevel08Decimal As Decimal = 985D**

**Const TaxLevel18Decimal As Decimal = 2450D**

**'Declare variables used within the function**

**Dim FederalTaxDecimal As Decimal**

**'Compute amount of tax based on the amount of pay**

**Select Case GrossPayDecimal**

**Case Is <= TaxLevel08Decimal '8% tax bracket**

**FederalTaxDecimal = Decimal.Round(TaxRate08Decimal \* GrossPayDecimal, 2)**

**Case Is <= TaxLevel18Decimal '18% tax bracket**

**FederalTaxDecimal = Decimal.Round(TaxRate18Decimal \* GrossPayDecimal, 2)**

**Case Else '28% tax bracket**

**FederalTaxDecimal = Decimal.Round(taxRate28Decimal \* GrossPayDecimal, 2)**

**End Select**

**'Return a value**

**Return FederalTaxDecimal**

**End Function**

**Private Function ComputeBenefitsDeduction(ByVal GrossPayDecimal As Decimal) As Decimal**

**'Benefit constants**

**Const MedicalRateDecimal As Decimal = 35.75D**

**Const LifeRateDecimal As Decimal = 18.35D**

**Const DentalRateDecimal As Decimal = 4D**

**'Declare variables used within the function**

**Dim BenefitsCostDecimal As Decimal**

**'Compute insurance benefits deduction**

**If MedicalCheckBox.Checked Then**

**BenefitsCostDecimal += MedicalRateDecimal 'selected medical insurance**

**End If**

**If LifeCheckBox.Checked Then**

**BenefitsCostDecimal += LifeRateDecimal 'selected life insurance**

**End If**

**If DentalCheckBox.Checked Then**

**BenefitsCostDecimal += DentalRateDecimal 'selected dental insurance**

**End If**

**'Use the retirement rate set in the CheckedChanged event**

**'for the retirement radio button controls**

**BenefitsCostDecimal += Decimal.Round(GrossPayDecimal \* RetirementRateDecimal, 2)**

**'Return cost of benefits**

**Return BenefitsCostDecimal**

**End Function**

**#End Region**

**End Class**