# Chapter 4

**Decisions and Conditions**

**Coverage:**

This chapter introduces the use of **If** statement for the evaluation of decisions in processing business data. You will learn various types of **If** statements including the **Block If**, **Nested If**, and **If-ElseIf-Else-End if** statements. You will also learn to use the **Case** structure with the **Select Case** statement to simplify decision logic. Other topics covered: relational operators, logical operators, testing control properties, using message boxes with multiple buttons, data validation, and debugging.

In this chapter you will build a project that computes payroll information for hourly employees of the VB University.

**Employee Information Groupbox** – application users type the employee name, ID, department (where the employee works), hours worked, and pay rate into the controls.

* The employee ID is a masked TextBox control – the social security number is the **Mask** property setting.
* For hourly employees, the hours worked (to the nearest 10th of an hour) and hourly pay rate are entered into two TextBoxes.
* The data entered must be numeric.
* The hours worked must be a number between 0 and 60 – the firm will not pay for more than 60 hours per pay period.

**Benefits Groupbox** – three radio button and three checkbox controls are used to capture information about employee benefit options.

* The selected retirement benefit defaults to **No Retirement Plan**.
* Employees have the option of not participating – this makes them responsible for setting up their own retirement plan—there is no deduction from the employee’s gross pay for the default “No Retirement Plan” option.
* Employees can opt for the firm’s standard plan – **5%** of their gross pay is deducted from their check and placed in a special retirement account – employees opting for this plan will be paid interest on their accumulated retirement deductions according to the T-bills market.
* Employees can participate in a 401A plan – **8%** of their gross pay is deducted from their check and the firm will match this up to $35 per pay period – the money is managed by a local firm that invests the money in mutual funds.
* Employees can optionally select any combination of insurance benefits.
* Medical insurance costs an hourly employee **$37.75** per pay period through a group health maintenance organization (HMO) plan.
* Life insurance is provided at a fee of **$18.35** per pay period through a special group employee plan.
* Dental insurance is **$4.00** per pay period.

**Payroll Information Groupbox** – four TextBox controls (with **ReadOnly** = **True**, **TabStop** = **False**, and **TextAlign** = **Right**) display output.

* **Gross Pay** – this is the pay rate times the hours work. Employees working over **40** hours per week are paid **1.5** times their normal pay rate for all hours worked over **40** hours.
* **Federal Tax** – the amount of federal tax due from an employee depends on the employee's gross pay amount. The federal tax structure is discussed later in these notes.
* **Benefits** – the cost of benefits (both retirement + insurance) selected by an employee is deducted from the gross pay.
* **Net Pay** – computed as **gross pay** minus **federal** **tax** minus **benefits**.

**Decision Structures and Commands**

**If Statements**

An **If** statement implements branching logic – use it when you need to execute a different set of program instructions depending upon some **condition** that can be tested. Here is an example:

* The left pane of this table shows what is termed **Pseudo Code** – a combination of VB and English to help you organize program logic.
* The right pane shows a **Program Flowchart** – a logic diagram that can also help you organize program logic.
* The **True** branch corresponds to the keyword **Then** and the **False** branch corresponds to the keyword **Else** in the pseudo code.
* The circle at the end represents the **End If** statement.
* You can code an unlimited number of statements inside either the **True** or **False** branches.

The general format of the **Block If** statement provides for alternative actions to take when the condition is **True** or **False**.

**If <condition> Then**

**(condition is true - do all of the actions**

**coded in this branch)**

**Else**

**(condition is false - do all of the actions**

**coded in this branch)**

**End If**

Learn these rules:

* A block **If** statement must always end with an **End If** statement.
* The key word **Then** must appear on the same line as the word If.
* The key words **Else** and **End If** must appear on lines by themselves.
* The **Else** branch is optional (see example below). Only use an **Else** statement if there is a **False** branch.
* Visual Basic automatically indents 4 spaces inside the **True** and **False** branches to make the statement easier to read.

**Decimal vs. Single vs. Double Data Types**

In this note set we use the **decimal** data type almost exclusively for data that includes a fixed decimal point – we apply this approach to both variable and constant values such as federal income tax rates, deductions for various benefits, etc. One might argue that the **single** or **double** data type is more appropriate. For example, the hours worked by an employee can be stored as either a single, double or decimal – these data types all allow for a decimal point and sufficient number of digits to the right of the decimal. These notes use **decimal** for the following reasons:

* It simplifies the coding because there is no need to convert a single or double data type to decimal when performing calculations such as multiplying hours worked by the worker’s pay rate.
* Hours worked is measured to the 1/10th of an hour – decimal data can store many significant digits to the right of the decimal point so no precision in data storage is lost.
* Several of the calculations are rounded using the **Decimal.Round** method that requires a decimal variable or expression (rather than a single or double) as the argument of the **Round** method.

**Conditions and Condition Symbols**

You must learn how to write conditions by using the relational operators shown in this table.

|  |  |
| --- | --- |
| **Relational   Operator** | **Meaning** |
| **>** | **greater than** |
| **<** | **less than** |
| **=** | **equal to** |
| **<>** | **not equal to** |
| **>=** | **greater than or equal to** |
| **<=** | **less than or equal to** |

* Conditions can be formed of numeric variables, constants, string variables, object properties, etc.
* Comparisons must be made comparing like data types to like data types, for example: Compare string to string (text), compare decimal to decimal, compare single to single.
* Numbers and expressions are compared using their mathematical value.
* Strings comparisons are covered later in these notes.

This is a list of sample **If** statements.

**If Decimal.Parse(AmountTextBox.Text) <= 400D Then**

**If MedicalCheckBox.Checked = True Then**

Is also equivalent to the following:

**If MedicalCheckBox.Checked Then**

**If DentalCheckBox.Checked <> False Then**

Is also equivalent to the following:

**If DentalCheckBox.Checked = True Then**

**If HoursDecimal > 60D Then**

**If TotalHoursInteger >= 100I Then**

**Single Condition If Statement**

The simplest **If** statement has no **Else** (False) branch, only a True branch -- no action is taken if the condition is false.

**If HoursDecimal <= 40D Then**

**'Pay only regular time**

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

**End If**

**Block If Statement with Else Branch**

The more common **If** statement has both **True** and **False** branches as shown in this example.

**If HoursDecimal <= 40D Then**

**'Pay only regular time**

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

**Else**

**'Pay regular time + overtime**

**GrossPayDecimal = Decimal.Round((40D \* PayRateDecimal) + \_**

**((HoursDecimal - 40D) \* PayRateDecimal \* 1.5D), 2)**

**End If**

Analyze the computation of gross pay for an overtime situation.

* The regular pay is computed with this part of the formula:

**(40D \* PayRateDecimal)**

* The hours worked over 40 hours is computed with this part of the formula:

**(HoursDecimal – 40D)**

* The overtime pay due is computed with this part of the formula:

**((HoursDecimal – 40D) \* PayRateDecimal \* 1.5D)**

* The **Decimal.Round** method rounds the gross pay to the nearest cent – the entire formula for gross pay computation is coded inside of the **Decimal.Round** method.

**VB Editor**

The VB Editor tries to help you write **If** statements.

* When in the code view window, if you type the keyword **If** and press Enter, the VB editor automatically adds the keywords **Then** and **End If**.
* You can add the **Else** branch as necessary.

The VB editor will try to correct errors.

* If you type **EndIf** with **no space**, the editor will correct this and add the required space.
* If you type **Else** with a coding statement on the line, the editor will add a **colon** between **Else** and the coding statement – the colon is a statement separator.
* These three coding segments show illegal syntax, VB editor’s correction, and the preferred syntax.

**Illegal Syntax**

**If HoursDecimal <= 40D Then**

**RegularPayCheckBox.Checked = True**

**Else RegularPayCheckBox.Checked = False**

**End If**

**VB Editor Correction with Colon**

**If HoursDecimal <= 40D Then**

**RegularPayCheckBox.Checked = True**

**Else : RegularPayCheckBox.Checked = False**

**End If**

**Preferred Syntax**

**If HoursDecimal <= 40D Then**

**RegularPayCheckBox.Checked = True**

**Else**

**RegularPayCheckBox.Checked = False**

**End If**

**If Statement – Comparing String Data**

Strings are compared beginning with the left-most character and are checked one character at a time based on the **ASCII** **code** (American Standard Code for Information Interchange) value of the character – ASCII is a subset of the **ANSI** **code** (American National Standards Institute).

* The ASCII table here shows codes 0 through 127.
* The ANSI extends the ASCII table to include codes 128 through 255 to code special characters (see the MSDN Help for the second chart).
* Characters are indicated under the **Char** column – their **decimal** and **hexadecimal** **value** (numbering system base 16) equivalents are shown in the **Dec** and **Hex** columns.
* The code column gives the ASCII equivalent of some special functions of the keyboard such as the Bell (Hex 07) and a carriage return (CR – Hex 0D).

ASCII Table (Codes 0 to 127)

This is like an alphabetic comparison, but strings may also contain special characters.

* All numbers that are part of strings are less than all letters (Hex values 48 – 57).
* A blank space is less than all numbers, characters, and special characters (Hex value 20).
* Some special characters are less than letters, but some are greater than letters.
* Upper-case letters (Hex values 41 – 5A) are less than lower-case letters (Hex values 61 – 7A).

VB stores string characters in **Unicode** – this coding system stores all characters as 2 bytes each to enable storage of all international languages (up to 65,536 unique character codes) – ANSI uses only the first byte.

**Which is Bigger?**

Analyze this coding segment and determine which message box will display.

**'Assign a value to two strings and compare them**

**Name1String = "Donnie"**

**Name2String = "Doug"**

**If Name1String > nameString2 Then**

**MessageBox.Show("Name: " & Name1String & " is bigger")**

**Else**

**MessageBox.Show("Name: " & Name2String & " is bigger")**

**End If**

The variable **NameString2** is actually bigger – a character-by-character comparison compares "**Don**" with "**Dou**" and since "**Don**" comes first in the alphabet, it is smaller, even though there are more characters in **Name1String** than in **Name2String**.

**ToUpper and ToLower Methods**

This example use the **ToUpper** method to treat the characters typed into a TextBox as if they are typed in upper-case, even though they may be typed in mixed case.

* This does not change the actual data stored, simply how the data values are treated.
* Makes it easier to determine if a value typed matches a name or address.

**If Name1String.ToUpper = "DOUG" Then**

**'Do some task involved in comparing values**

**End If**

The **ToLower** method is the opposite of **ToUpper** – all characters are treated as if they are typed in lower-case letters.

**Logical Operators**

Logical operators are used to combine conditions. These are termed **compound conditions**. The logical operators are as follows:

* **Or** operator – If one condition or both conditions are **True**, the entire compound condition evaluates to **True**. In this example, if a checkbox control named **SalaryRadioButton** is checked or a second checkbox control named **ContractRadioButton**, then the worker is paid on a monthly basis.

**If SalaryRadioButton.Checked = True Or ContractRadioButton.Checked = True Then**

**'Worker is paid a monthly salary**

**Else**

**'Worker is paid an hourly wage**

**End If**

* **And** operator – Both conditions being combined must be **True** in order for the compound condition to be **True**. In this example, if a TextBox control named **HoursTextBox** enabled **AND** if the data in **HoursTextBox** is not numeric data, then this is a data error condition that must be corrected before data can be processed.

**If HoursTextBox.Enabled = True And IsNumeric(HoursTextBox.Text) = False Then**

**'Hours must be numeric – display error message**

**Else**

**'Process the good data**

**End If**

* **Not** operator – Negates a condition – if a condition is **True**, it is treated as **False**, and vice-versa. In this example, an existence test is made for the existence of data in a TextBox control.

**If Not NameTextBox.Text = String.Empty Then**

**'Name is not missing - process the data**

**Else**

**'Name cannot be missing - display error message**

**End If**

The **Not** operator can be confusing. You can rewrite the above coding segment by using the not equal to comparison operator:

**If NameTextBox.Text <> String.Empty Then**

**'Name is not missing - process the data**

**Else**

**'Name cannot be missing - display error message**

**End If**

You can also use reverse logic to rewrite the coding segment.

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

**'Name cannot be missing - display error message**

**Else**

**'Name is not missing - process the data**

**End If**

* **AndAlso** operator – Termed a short-circuit of the **And** operator.
  + If the first condition is **False**, the compound condition is treated as **False** and the second condition is not evaluated.
  + In this example, if the value in the **HoursTextBox** TextBox is **not numeric**, then there is no need to test the second condition – in fact, if the value is not numeric, testing the value to see if it falls within a numeric range would lead to an **exception error** due to trying to compare non-numeric data with numeric values.

**If IsNumeric(HoursTextBox.Text) = True AndAlso \_**

**Decimal.Parse(HoursTextBox.Text, \_**

**Globalization.NumberStyles.Number) <= MaxHoursDecimal Then**

**'The data is valid, use this branch of the**

**'decision structure to process the data**

**Else**

**'Data is not valid, hours must be numeric and within allowable range**

**MessageBox.Show("Hours worked must be less than " & MaxHoursDecimal.ToString, "Hours Numeric Error", MessageBoxButtons.OK, MessageBoxIcon.Error)**

**HoursTextBox.Focus()**

**HoursTextBox.SelectAll()**

**End If**

* **OrElse** operator – Termed a short-circuit of the **Or** operator.
  + If the first condition is **True**, the compound condition is treated as **True** and the second condition is not evaluated.
  + This is sort of a mirror-image of the **AndAlso** operator – sometimes you will use one, sometimes the other depending upon how you choose to structure your logic.
  + In this example, the data in the **HoursTextBox** TextBox control is first evaluated to see if it is numeric – if it is not then the condition evaluates to **True** – since Not Numeric is False the second condition testing if the hours worked is less than **0D** or greater than **MaxHoursDecimal** is not tested.
  + Note that the two conditions **cannot** be reversed – you must test for numeric data before testing to see if data is less than or equal to zero or greater than some maximum value because if the data is **not numeric**, then the second condition would cause an exception – you cannot compare non-numeric data to a number.

**If IsNumeric(HoursTextBox.Text) = False OrElse \_**

**Decimal.Parse(HoursTextBox.Text, \_**

**Globalization.NumberStyles.Number) > MaxHoursDecimal Then**

**'Data is not valid - hours must be numeric and within allowable range**

**MessageBox.Show("Hours worked must be a number between 0 and " & MaxHoursDecimal.ToString, Hours Numeric Error", MessageBoxButtons.OK, MessageBoxIcon.Error)**

**HoursTextBox.Focus()**

**HoursTextBox.SelectAll()**

**Else**

**'The data is valid, use this branch of the**

**'decision structure to process the data**

**End If**

* **Xor** operator – Termed the **Exclusive Or** operator.
  + If one condition is **True**, the other must be **False** in order for the compound condition to be treated as **True** overall.
  + Both conditions cannot be **True**.
  + In this example, if a big spender is in the club, then the club will be closed to further customers, or if the number of customers is greater than 250 the club is closed to further customers; however, we cannot have the club full of customers **and also** attract a big spender.
  + The **Xor** operator is rarely used in business applications—it is only useful where situations are mutually exclusive.

**If BigSpenderRadioButton.Checked = True Xor \_**

**NumberCustomersInteger > 250 = True Then**

**'The club is full enough – lock the door**

**Else**

**'Continue to accept customers in the door**

**End If**

This table summarizes the results of using the And, Or, AndAlso, OrElse, and Xor logical operators to evaluate a compound condition that is comprised of two simple conditions. It also summarizes the effect of the Not operator on any condition.

|  |  |  |  |
| --- | --- | --- | --- |
| **Logical**  **Operator** | **Simple**  **Condition 1** | **Simple**  **Condition 2** | **Compound**  **Condition Value** |
| And | True  True  False  False | True  False  True  False | True  False  False  False |
| Or | True  True  False  False | True  False  True  False | True  True  True  False |
| AndAlso | True  True  False | True  False  Not Evaluated (short circuits) | True  False  False |
| OrElse | True  False  False | Not Evaluated (short circuits)  True  False | True  True  False |
| Xor | True  True  False  False | True  False  True  False | False  True  True  False |
| Not | True  False | --  -- | False  True |

**In-Class Exercise**

You will not build the form for the project in class; rather, you should copy the **Ch04VBUniveristy-Start Project** folder from the drive Y: server directory for CMIS 142. You can elect to practice building the form on your own, but you will not do this in class.

**Compute Button Click Event Sub Procedure**

This is a very complex sub procedure that requires the use of multiple **If** statements and assignment statements in order to compute and display the correct output.

The pseudo code for the sub procedure is shown at a very high-level here in the form of remarks statements. Each of the remarks requires multiple lines of code in order to accomplish the task.

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

**Try**

**'Parse TextBox values to memory variables**

**'Compute gross pay**

**'Compute federal tax**

**'Compute insurance benefits deduction**

**'Compute retirement benefits deduction**

**'Compute net pay**

**'Display output**

**Catch ex As Exception**

**'Display generic error message**

**End Try**

**End Sub**

**Compute Gross Pay**

The pseudo code shown here includes parsing TextBox values and computing gross pay is:

**'Convert TextBox values for hours worked and pay rate to memory**

**'If hours worked <= 40 Then**

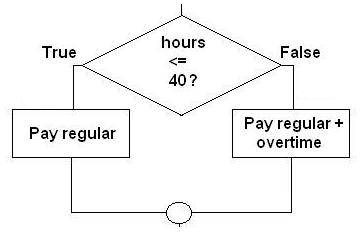
**' pay only regular time**

**'Else**

**' pay regular time + overtime**

**'End If**

* This computation uses a standard block **If** statement with both **True** and **False** branches.
* A flowchart diagram for this logic is shown here.



* There is more than one way to organize this logic – this example reverses the first **If** statement to test for hours > **40**. Both approaches are correct – how you code the problem depends on how you logically organize the **If** statement conditions.

**'Try**

**' Convert TextBox values for hours worked and pay rate to memory**

**' If hours worked > 40 Then**

**' pay regular time + overtime**

**' Else**

**' pay only regular time**

**' End If**

**'Catch**

**' MessageBox.Show("Unexpected Error")**

**'End Try**

* Use the first pseudo code shown above and convert it to VB code. The solution is shown here but the **Try-Catch** block is not shown. Remember to include the Try-Catch block in your code.

**'Earlier in the sub procedure the memory variables must be declared**

**'Declare variables**

**Dim HoursDecimal, PayRateDecimal, GrossPayDecimal As Decimal**

**'Parse TextBox values to memory variables**

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

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

**'Compute gross pay**

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

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

**Else 'pay regular + overtime**

**GrossPayDecimal = Decimal.Round((40D \* PayRateDecimal) \_**

**+ ((HoursDecimal - 40D) \* PayRateDecimal \* 1.5D), 2)**

**End If**

**'Later in the sub procedure**

**'Display output**

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

* Code the Catch block.

**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**

* Prior to computing the value for **GrossPayDecimal** (a memory variable), the **Text** property of the hours worked and pay rate TextBox controls are parsed and assigned to memory variables.
* You can optionally decide to declare constants for the numeric values **40D** and **1.5D**; however, these are fixed in our society by law and so coding these values as constants is probably not necessary.

**Testing the Program**: – requires a setting break point.

* Set a break point at the beginning of the Compute Button’s sub procedure.
* Run the program and enter data for which the results will be known, e.g., hourly worker working 40 hours @ $10/hour.
* Click the Compute button.
* Press F8 to step through the code line by line.
* Place the mouse pointer over a variable name or control property to examine the current value at the time of execution and VB displays the current value of the variable.

**Compute Federal Tax – Using If-ElseIf-Else-End If Logic**

The simplified tax table shown here is used to compute federal taxes. The tax rates are 8%, 18%, and 28%.

|  |  |
| --- | --- |
| **Federal Tax Rates** | |
| **Gross Pay** | **Tax Rate** |
| Up to $985 per pay period | 0.08 |
| $985.01 - $2,450.00 per pay period | 0.18 |
| Above $2,450.01 per pay period | 0.28 |

The logic for computing taxes can be implemented as shown in this logic diagram.

* If the gross pay is less than **$985**, then tax is computed at the **8%** rate and control passes out of the structure.
* If gross pay is more than **$985** but less than **$2,450**, then tax is computed at the **18%** rate.
* The last **False** branch has tax computed at the **28%** rate.

This can be coded with multiple **If** statements as shown here.

* Declare tax rate constants to store the tax rate as well as the break points (income levels) as shown here. These can be coded as either **module-level constants** or as **local constants** within the Compute button's click event. We will code these as local constants as they are only used within this single sub procedure.
* Declare a variable to store the amount of federal tax due, **FederalTaxDecimal** (highlighted in yellow).

**'Declare variables and constants**

**Dim HoursDecimal, PayRateDecimal, GrossPayDecimal, FederalTaxDecimal As Decimal**

**'Tax rate constants**

**Const TAX\_RATE\_08\_DECIMAL As Decimal = 0.08D**

**Const TAX\_RATE\_18\_DECIMAL As Decimal = 0.18D**

**Const TAX\_RATE\_28\_DECIMAL As Decimal = 0.28D**

**Const TAX\_LEVEL\_08\_DECIMAL As Decimal = 985D**

**Const TAX\_LEVEL\_18\_DECIMAL As Decimal = 2450D**

The **If** statement coding is:

**. . . inside the Compute button click event**

**'Compute federal tax**

**If GrossPayDecimal <= TAX\_LEVEL\_08\_DECIMAL Then '8% tax bracket**

**FederalTaxDecimal = Decimal.Round(TAX\_RATE\_08\_DECIMAL \* GrossPayDecimal, 2)**

**ElseIf GrossPayDecimal <= TAX\_LEVEL\_18\_DECIMAL Then '18% tax bracket**

**FederalTaxDecimal = Decimal.Round(TAX\_RATE\_18\_DECIMAL \* GrossPayDecimal, 2)**

**Else '28% tax bracket**

**FederalTaxDecimal = Decimal.Round(TAX\_RATE\_28\_DECIMAL \* GrossPayDecimal, 2)**

**End If**

**'Later in the sub procedure**

**'Display output**

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

Add the code required to compute and display federal income tax to your program.

* Add the module-level constants.
* Add **FederalTaxDecimal** to the Dim statement inside the sub procedure used to declare decimal variables.
* Add the code to compute federal tax due from the employee after computing the gross pay.
* Add a line of code to produce formatted output displayed to the **FederalTaxTextBox** control.
* Test the program.

**Test the program:** Run the program and enter data to test each tax bracket.

* 8% bracket – use 40 hours times $20/hour. Tax due: $64.00.
* 18% bracket – use 40 hours times $25/hour. Tax due: $180.00.
* 28% bracket – use 40 hours times $62/hour. Tax due: $694.40.

**Compute Federal Tax – Using a Select Case Structure**

The **Select Case** block statement in Visual Basic is a form of decision structure. It can be used in place of an **If** statement when a single variable value, expression value, control property, or similar object is to be evaluated and different actions are taken depending on the value of the expression.

You can use the **Select Case** statement to compute the federal income tax due instead of using an **If** statement. The advantage of the **Select Case** of multiple **If** statements is that the **Select Case** is often easier to read and code than multiple If statements.

The **Select Case** coding approach is shown here. The **Case Else** branch is optional and covers all other case values listings.

**. . . inside the Compute button click event**

**'Compute federal tax**

**Select Case GrossPayDecimal**

**Case Is <= TAX\_LEVEL\_08\_DECIMAL '8% tax bracket**

**FederalTaxDecimal = Decimal.Round(TAX\_RATE\_08\_DECIMAL \* GrossPayDecimal, 2)**

**Case Is <= TAX\_LEVEL\_18\_DECIMAL '18% tax bracket**

**FederalTaxDecimal = Decimal.Round(TAX\_RATE\_18\_DECIMAL \* GrossPayDecimal, 2)**

**Case Else '28% tax bracket**

**FederalTaxDecimal = Decimal.Round(TAX\_RATE\_28\_DECIMAL \* GrossPayDecimal, 2)**

**End Select**

There are several rules to learn about coding a **Select Case**:

* When using a comparison operator such as **=** or **<=** the keyword **Is** must be used; examples:

**Case Is <= TAX\_LEVEL\_08\_DECIMAL**

**Case Is >= 2580**

Compare the **Select Case** to the equivalent **If** statement code:

**If ValueDecimal < 150 Then**

**'Do one thing**

**ElseIf ValueDecimal >= 2580 Then**

**'Do a second thing**

**Else**

**'Do some alternative thing**

**End If**

**Select Case ValueDecimal**

**Case Is < 150**

**'Do one thing**

**Case Is >= 2580**

**'Do a second thing**

**Case Else**

**'Do some alternative thing**

**End Select**

* To test for a value that falls within a range of constants, the keyword **To** is used; example:

**Case 25 To 72**

**Case 90 To 100**

Compare the **Select Case** to the equivalent **If** statement code:

**If ValueDecimal >= 25 And ValueDecimal <= 72 Then**

**'Do one thing**

**ElseIf ValueDecimal >= 90 And ValueDecimal <= 100 Then**

**'Do a second thing**

**Else**

**'Do some alternative thing**

**End If**

**Select Case ValueDecimal**

**Case 25 To 72**

**'Do one thing**

**Case 90 To 100**

**'Do a second thing**

**Case Else**

**'Do some alternative thing**

**End Select**

* A list can combine individual values and ranges of values; example:

**Case 15, 17, 25 To 72, 79, Is > 150**

Compare the **Select Case** to the equivalent **If** statement code:

**If ValueDecimal = 15 Or ValueDecimal = 17 Or \_**

**(ValueDecimal >= 25 And ValueDecimal <= 72) Or \_**

**ValueDecimal = 79 Or ValueDecimal > 150 Then**

**'Do something**

**Else**

**'Do alternative**

**End If**

**Select Case ValueDecimal**

**Case 15, 17, 25 To 72, 79, Is > 150**

**'Do something**

**Case Else**

**'Do alternative**

**End Select**

* To test a string value, include the literal value within double-quote marks; example:

**Select Case TeamNameTextBox.Text.ToUpper**

**Case "RAMS", "PATRIOTS"**

**'Code to prepare for Super bowl**

**Case "RAIDERS", "COWBOYS"**

**'Code to prepare for playoffs**

**Case Else**

**'Code to prepare for long break**

**End Select**

Replace the tax computation code with the **SELECT CASE** code above for computing federal income tax to your program. Test the program again.

**Test the program:** Run the program and enter data to test each tax bracket.

* 8% bracket – use 40 hours times $20/hour. Tax due: $64.00.
* 18% bracket – use 40 hours times $25/hour. Tax due: $180.00.
* 28% bracket – use 40 hours times $62/hour. Tax due: $694.40.

**Compute Insurance Benefit Cost**

The cost of insurance benefits is determined by testing the **Checked** property of the three insurance benefit checkbox controls. Any combination of the three controls may be checked: 0, 1, 2, or all 3.

You might decide to use the **And** logical operator to test the various checkbox combinations; unfortunately, this approach becomes infeasible when the number of check box controls is large. It is better to use simple **If** statements as shown in the solution given below.

* Declare the **BenefitsCostDecimal** variable by adding it to the **Dim** statement declaring all decimal variables for the sub procedure. This is an accumulating variable within the sub procedure so it does not need to be module-level.
* The cost of each insurance plan on a per pay period basis is stored as a constant – declare the constants as local or module-level, it does not matter – the example code uses local in because the constants are only used in this single sub procedure.
* Note the use of the **+=** (plus equal) operator – this causes the benefit cost to accumulate.

**'Declare variables and constants**

**Dim HoursDecimal, PayRateDecimal, GrossPayDecimal, FederalTaxDecimal, BenefitsCostDecimal As Decimal**

**. . .**

**'Benefit constants**

**Const MEDICAL\_RATE\_DECIMAL As Decimal = 35.75D**

**Const LIFE\_RATE\_DECIMAL As Decimal = 18.35D**

**Const DENTAL\_RATE\_DECIMAL As Decimal = 4D**

**. . . inside the Compute button click event**

**'Compute insurance benefits deduction**

**If MedicalCheckBox.Checked Then**

**BenefitsCostDecimal += MEDICAL\_RATE\_DECIMAL 'selected medical insurance**

**End If**

**If lifeCheckBox.Checked Then**

**BenefitsCostDecimal += LIFE\_RATE\_DECIMAL 'selected life insurance**

**End If**

**If DentalCheckBox.Checked Then**

**BenefitsCostDecimal += DENTAL\_RATE\_DECIMAL 'selected dental insurance**

**End If**

**'Later in the sub procedure**

**'Display output**

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

**Test the program:** Run the program and enter data to test each insurance benefit.

**Compute Retirement Benefit Cost**

The cost of the retirement benefit selected is determined by testing the **Checked** property of the three retirement benefit radio button controls. Unlike checkboxes, with radio buttons, only one of the controls can be selected.

Here you may be tempted to try to use the **Or** logical operator to test the various combinations, but this approach is awkward and can become infeasible when the number of radio button controls is large.

The best approach uses the **If-ElseIf-Else-End If** decision structure since only one of the conditions (radio button checked) will be true.

* The same **BenefitsCostDecimal** variable is used to add the cost of insurance benefits to the already accumulated cost of insurance benefits to arrive at a total cost of benefits (insurance + retirement) within the sub procedure.
* The cost of the standard and 401A retirement plans on a per pay period basis is stored as constants, either local module-level, it does not matter. Here just to demonstrate the use of module-level constants the constants are coded as module-level.
* Note that since the cost of no retirement plan is zero, this doesn’t need to be coded.
* The use of the **+=** (plus equal) operator causes the cost to accumulate.
* The cost of the retirement benefit is computed by multiplying the retirement percentage by the gross pay value, and is rounded to the nearest penny.

**'Module-level variable and constant declarations**

**. . .**

**'Benefit constants**

**. . .**

**Const RETIREMENT\_STANDARD\_DECIMAL As Decimal = 0.05D**

**Const RETIREMENT\_401A\_DECIMAL As Decimal = 0.08D**

**. . . inside the Compute button click event**

**'Compute retirement benefits deduction**

**If Retirement401ARadioButton.Checked Then**

**BenefitsCostDecimal += Decimal.Round(RETIREMENT\_401A\_DECIMAL\* GrossPayDecimal, 2)**

**ElseIf RetirementStandardRadioButton.Checked Then**

**BenefitsCostDecimal += Decimal.Round(RETIREMENT\_STANDARD\_DECIMAL \* GrossPayDecimal, 2)**

**Else**

**'No charge for not taking retirement benefit**

**End If**

**'Later in the sub procedure**

**'Display output**

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

**Test the program:** Run the program and enter data to test each retirement benefit.

**Compute Net Pay and Display Output (Formatted)**

The net pay computation is a straight-forward formula: **net pay = gross pay – taxes – benefits**

The solution is shown below along with the additional assignment statement to display the net pay. Remember to add the **NetPayDecimal** variable to the **Dim** statement declaring all decimal variables for the sub procedure.

**'Declare variables and constants**

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

**. . .**

**'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("C")**

**federalTaxTextBox.Text = FederalTaxDecimal.ToString("N")**

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

**netPayTextBox.Text = NetPayDecimal.ToString("C")**

**Test the program:** Run the program and enter data to test each to confirm that the net pay computation is correct – use a calculator to validate the solution.

**Exit Button Click Event Sub Procedure – Using Multiple MessageBox Buttons**

Sometimes an application user may accidentally exit an application. This can be prevented with the code solution given here:

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

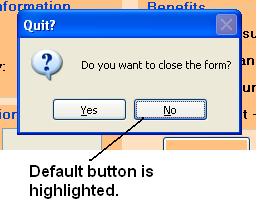
**If ButtonDialogResult = Windows.Forms.DialogResult.Yes Then**

**Me.Close()**

**End If**

**End Sub**

* A string variable named **MessageString**  is declared – it stores the question to be asked to the system user when the **Exit** button is clicked.
* A dialog result variable named **ButtonDialogResult** is declared and set equal to the **MessageBox.Show** method – dialog result variables store values representing a dialog interface with the application user.
* Program execution pauses while a **MessageBox** displays with the message, an appropriate title bar and two buttons (**Yes** and **No**).



* The **MessageBoxDefaultButton.Button2** parameter specifies that the **No** button is the default if the enter key is pressed – this will keep the form from closing accidentally.
* The **If** statement compares the value stored in **ButtonDialogResult** to the **Windows.Forms.DialogResult.Yes** enumerated value (recall that enumerated values are VB intrinsic constants defined for your ease of use).

**Test the program:** Add the code to your project and test the **Exit** button.

**Input Data Validation**

**Validating Event and ErrorProvider Control**

The application user is likely to make occasional data entry errors. A form’s data must be validated against a list of **business rules** developed by a systems analyst. Example business rules include:

* **Missing Data Test** – data values cannot be missing – means TextBox controls cannot be empty.
* **Numeric Data Test** – numeric values must be tested to ensure they are numeric.
* **Reasonableness Test** – values (usually numeric) must fall within a specified reasonable range.

In this note set we use the **Validating** event for a control such as a TextBox to validate the contents of the control.

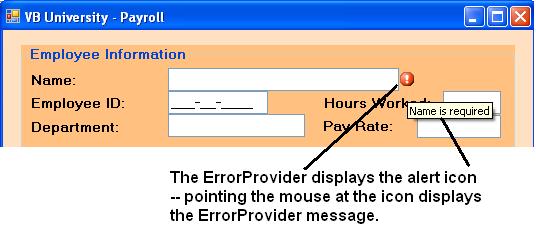
* The **Validating** event fires when you click away from a control with the mouse or when you tab away from a control – basically when a control loses **focus**.
* If a control never gets the **focus**, then the **Validating** event for that control **does not fire**.
* If a control fails validation, you can make the control to retain focus by setting the **e.Cancel** property = **True** – focus returns to the control so the invalid data can be corrected.

What if you do **NOT** want to validate data for a control?

* Set the control’s **CausesValidation** property = **False** (the default is **True**) – this causes the control to **not** trigger validation.
* Example: You want to exit a form when clicking the **Exit** Button control – or you want to reset a form without validating when you click the **Reset** Button control – set the **CausesValidation** = **False** in the properties window at design mode so the control does not cause a **Validating** event to fire.

An **ErrorProvider** control displays an **alert icon** and an **error message** next to a control that contains invalid data as shown in the figure below.

* The **ErrorProvider** control is found in the **Toolbox-Components** section.
* The **ErrorProvider** control displays in the system component tray.
* The default name is **ErrorProvider1** – we will use this name as there is no need to rename the control.
* A form only needs one **ErrorProvider** control.



The logic we will use for a **Validating** event with the **ErrorProvider** control is illustrated in this figure.

* Use **Negation logic** by testing to determine if the business rule is violated (data is not valid).
* If data is not valid:
  + **e.Cancel** = **True** (this cancels the control losing focus)
  + Set the **ErrorProvider** message for the control with invalid data to an appropriate message
* Else
  + Clear the **ErrorProvider** message for the control so it is blank (empty string).
* End If

**Missing Data Test – Employee Name**

A **Missing Data** validation test is also called an **Existence Test** or **Required Field Test** – you must test a control such as a TextBox to ensure that data that must be is not missing.

* The **NameTextBox** control requires a name in order to process payroll.
* If the name is missing, you use an **ErrorProvider** control to inform the application user that the data is missing.
* Access the **Validating** event in the coding window by using the two drop-down options – select the control name to be validated from the first drop-down, then select the Validating event from the second dropdown (labeled Declarations).

**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**

* The above sub procedure uses an **If** statement to compare the **Text** property of the NameTextBox control to the VB enumerated value **String.Empty**.
* If the TextBox is empty, the **e.Cancel** property of the **CancelEventArgs** object is set to **True** – this cancels leaving the TextBox and causes the TextBox to keep the focus.
* The **SetError** method of the **ErrorProvider** is set to the control that has the error and the error message is specified in double-quotes.
* If the TextBox data is valid (not empty), the **ErrorProvider** message is set to the empty string by using two double-quotes with nothing between the double-quotes.

A **MessageBox** control can also be used with a Validating event; however, it does not work as well as an **ErrorProvider** – for this reason, we will always use the **ErrorProvider** with a **Validating** Event.

**Missing Data Test – Employee ID and Department**

Both the EmployeeID and Department are required data values. These are additional examples of missing data validation.

The **EmployeeIDMaskedTextBox**  control has the **Mask** property set to the **SSN** mask.

* If no **SSN** is entered the **Text** property of the control stores the value **space-space-space-dash-space-space-dash**. The four trailing spaces are not stored.
* The rest of the code works as it did for the **NameTextBox** control.

**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**

The **DepartmentTextBox** control is validated in the same fashion as the **NameTextBox** control.

**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**

**Numeric Data Test and Reasonableness Test – Hours Worked and Pay Rate**

A TextBox value that must be numeric is tested with the **IsNumeric** function.

* Hours worked must be numeric and also fall within a specific valid range – **zero to 60** hours.
* This uses an **OrElse** short-circuit logical operator.
* The first part of the compound condition uses the **IsNumeric** function to determine if the hours worked is a valid number (highlighted in yellow).
* The second half of the compound condition tests if the parsed value of the TextBox is **<= 0** or **>** **MaxHoursDecimal** (highlighted in light blue). The **MaxHoursDecimal** constant stores the value **60**.

**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**

An alternative to using the **OrElse** operator is to break the compound condition into two separate, nested If statements.

* The first **If** statement tests for a numeric value – if it is not numeric, **e.Cancel** retains the control’s focus and a specific error message is given.
* The nested second **If** statement only executes if hours worked is a number – this tests that the number is within the value range **0 to 60**.
* This logic is considerably more complex – **key learning point:** learn to use the **OrElse** operator.

**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 Then**

**'Hours must be numeric**

**e.Cancel = True**

**ErrorProvider1.SetError(HoursTextBox, "Hours worked must be a number")**

**Else**

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

**'Hours must be within allowable range**

**e.Cancel = True**

**Dim errorMessageString = "Hours worked must be between 0 and " & MaxHoursDecimal.ToString**

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

**HoursTextBox.SelectAll()**

**Else**

**'Clear error provider message**

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

**End If**

**End If**

**End Sub**

The **PayRateTextBox** control must also be validated for numeric values.

* This also uses the **OrElse** operator.
* The reasonableness test here is simpler – the pay rate is only invalid if it is **<= 0**.

**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**

**Overriding Validation – Closing a Form**

One problem with the **Validating** event is that when a control contains invalid data, it is impossible to close the form by clicking the **Close box** or by calling the **Close** method in code.

* Override validation to close a form by coding the **FormClosing** event.
* The code is straight-forward. Use the **e.Cancel** method.

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

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

**e.Cancel = False**

**End Sub**

**Test the program:** Add the code to your program.

* Paste the code for the **Validating** events and the **FormClosing** event into your program.
* Test each business rule.

**Reset Button Click Event Sub Procedure**

The code to reset the form is shown here. You should understand this code from your study of earlier chapters.

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

**'Set focus to name textbox**

**NameTextBox.Focus()**

**End Sub**

**Combining Radio Button CheckedChanged Events**

An alternative way to program the cost of a benefit that is represented by **radio button controls** is through the **CheckedChanged** event. Each radio button control can have a **CheckedChanged** event, or a **group** of radio button controls can share the same **CheckedChanged** event as is demonstrated here.

* A module-level variable is created to store the cost of the retirement benefit.

**'Variable to store rate of retirement benefit**

**Private RetirementRateDecimal As Decimal**

* A **CheckedChanged** event is coded for one of the radio buttons (the button selected does **NOT** matter).
* The **Handles** clause is modified to handle the event for each of the three retirement event radio buttons (highlighted in yellow).

**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 RETIREMENT\_STANDARD\_DECIMAL As Decimal = 0.05D**

**Const RETIREMENT\_401A\_DECIMAL 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 = RETIREMENT\_STANDARD\_DECIMAL**

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

**RetirementRateDecimal = RETIREMENT\_401A\_DECIMAL**

**End Select**

**End Sub**

* The **Handles** clause is modified to handle the **CheckedChanged** event for each of the status radio button controls as highlighted above in yellow.
* You can simply type a **comma** at the end of an existing **Handles** clause and use **Intellisense** to access the control/event combination that you also want this sub procedure to handle.
* It does not matter which sub procedure you select initially – here the **NoneRadioButton** radio button’s **CheckedChanged** event sub procedure was selected.
* A "**generic**" radio button object is created in memory with the **Dim** statement shown here.
* The **CType** (convert type) function is used to convert the **sender** object, which represents the radio button selected by the application user from an "**object**" to a "**radio button**" control and store it to the radio button object that exists in memory – this step is necessary because the **sender** object does not actually exist until run time – trying to reference **sender** as an object prior to this will result in an exception message known as "**late binding not allowed**".
* Since radio buttons have properties, these properties can be evaluated.
* The **Name** property of the memory radio button is evaluated to determine which button has the **Checked** property changed.
* Each case sets the **RetirementRateDecimal** module-level variable to the percent of gross pay to be deducted from the employee’s pay check by storing a module-level constant with the correct % value to the variable.

**Modify the program:** Prior to testing you must complete these tasks:

* Declare the **RetirementRateDecimal** module-level variable.

**Public Class Payroll**

**'Module level variable/constant declarations**

**Private RetirementRateDecimal As Decimal**

* Add the **CheckedChanged** event as shown above.
* Modify the Compute button control’s Click Event by **Remarking** out the code that accumulates the cost of the retirement benefit (the **If-ElseIf-Else-End If** code), and by deleting the declarations for the two retirement rate constants.

**'Remark out this part to test use of the CheckedChanged event to**

**'set the retirement rate**

**''Compute retirement benefits deduction**

**'If Retirement401ARadioButton.Checked Then**

**' BenefitsCostDecimal += Decimal.Round(RETIREMENT\_401A\_DECIMAL\* GrossPayDecimal, 2)**

**'ElseIf RetirementStandardRadioButton.Checked Then**

**' BenefitsCostDecimal += Decimal.Round(RETIREMENT\_STANDARD\_DECIMAL \* GrossPayDecimal, 2)**

**'Else**

**' 'No charge for not taking retirement benefit**

**'End If**

* Add a single line of code as shown here to accumulate the cost of the retirement benefit.

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

**'for the retirement radio button controls**

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

**Test the program:** Confirm that the cost of retirement benefit is still computed correctly.

**Debugging Visual Basic**

The various debugging tools can help find logic and run-time errors.

**Debug Output and the Immediate Window**

The **immediate window** can display output during program execution for debugging purposes.

* Use the **Debug.WriteLine** method to write a value to the immediate window – this is useful when you are working through a program line-by-line.
* Example: Insert this code at the beginning of the Compute button's click sub procedure to write information to the immediate window:

**Debug.WriteLine("Started Compute Button Click Event")**

* Clear the immediate window by right-clicking within the immediate window and choosing **Clear All**.

**Break All**

Clicking the **Break All** icon on the **Debug** toolbar will cause execution to break when running a program and will place you in **debug** **mode**. I haven't found this to be particularly useful – it is often better to force a break point – it can be used when the program is stuck in an endless loop.

**Break Points**

Earlier in these notes you learned how to set a **break point** for a sub procedure – the Compute Button's click event sub procedure – by clicking in the gray vertical bar on the left side of the coding window (see the figure below).

* A break point can be set on any executable line of code.
* If you know that code is executing satisfactorily up to a particular point in a sub procedure, then set the break point where you need to enter debug mode as shown in this figure.
* As you learned earlier, you can check the current value of a variable or property (or any expression) by putting the mouse pointer over the object.
* Use the **Step Over** menu option (or debug toolbar icon or **F8**) to step line by line.
* Use the **Step Out** (or debug toolbar icon or **Ctrl+Shift+F8**) to finish execution of the current procedure.
* Use the **Stop Debugging** option to end the run when you find the error.
* Use the **Continue option** to run the program to the next natural break in execution where the program waits for additional input or application actions.
* You can toggle break points or clear all break points from the **Debug** menu.
* **Edit and Continue** – During **Debug** mode, you can modify code to correct errors, then press **F5** (or Debug/Continue) to continue code execution for testing purposes.

**Locals Windows**

This window displays the values of all objects and variables that are within **local** **scope** at break time.

* Expand the **Me** entry to see the state of all of the form's controls.
* Note the current values being modified are highlighted in red.

**Autos Window**

This window displays all variable and control contents referenced in the current statement and **three statements** before/after the current statement.

**Solution**

'Project: Ch04VBUniversity (Solution)

'CS2

'Today's Date

Public Class Payroll

'Module level variable/constant declarations

'Declare retirement benefit constants

Const RETIREMENT\_STANDARD\_DECIMAL As Decimal = 0.05D

Const RETIREMENT\_401A\_DECIMAL As Decimal = 0.08D

Private RetirementRateDecimal As Decimal

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

Try

'Declare variables and constants

Dim HoursDecimal, PayRateDecimal, GrossPayDecimal, FederalTaxDecimal, BenefitsCostDecimal, NetPayDecimal As Decimal

'Tax rate constants

Const TAX\_RATE\_08\_DECIMAL As Decimal = 0.08D

Const TAX\_RATE\_18\_DECIMAL As Decimal = 0.18D

Const TAX\_RATE\_28\_DECIMAL As Decimal = 0.28D

Const TAX\_LEVEL\_08\_DECIMAL As Decimal = 985D

Const TAX\_LEVEL\_18\_DECIMAL As Decimal = 2450D

'Benefit constants

Const MEDICAL\_RATE\_DECIMAL As Decimal = 35.75D

Const LIFE\_RATE\_DECIMAL As Decimal = 18.35D

Const DENTAL\_RATE\_DECIMAL As Decimal = 4D

'Parse textbox values to memory variables

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

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

'Compute gross pay

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

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

Else 'pay regular + overtime

GrossPayDecimal = Decimal.Round((40D \* PayRateDecimal) \_

+ ((HoursDecimal - 40D) \* PayRateDecimal \* 1.5D), 2)

End If

'Compute federal tax

Select Case GrossPayDecimal

Case Is <= TAX\_LEVEL\_08\_DECIMAL '8% tax bracket

FederalTaxDecimal = Decimal.Round(TAX\_RATE\_08\_DECIMAL \* GrossPayDecimal, 2)

Case Is <= TAX\_LEVEL\_18\_DECIMAL '18% tax bracket

FederalTaxDecimal = Decimal.Round(TAX\_RATE\_18\_DECIMAL \* GrossPayDecimal, 2)

Case Else '28% tax bracket

FederalTaxDecimal = Decimal.Round(TAX\_RATE\_28\_DECIMAL \* GrossPayDecimal, 2)

End Select

'Compute insurance benefits deduction

If MedicalCheckBox.Checked Then

BenefitsCostDecimal += MEDICAL\_RATE\_DECIMAL 'selected medical insurance

End If

If LifeCheckBox.Checked Then

BenefitsCostDecimal += LIFE\_RATE\_DECIMAL 'selected life insurance

End If

If DentalCheckBox.Checked Then

BenefitsCostDecimal += DENTAL\_RATE\_DECIMAL 'selected dental insurance

End If

''Remark out this part to test use of the CheckedChanged event to

''set the retirement rate

''Compute retirement benefits deduction

'If Retirement401ARadioButton.Checked Then

' BenefitsCostDecimal += Decimal.Round(RETIREMENT\_401A\_DECIMAL \* GrossPayDecimal, 2)

'ElseIf RetirementStandardRadioButton.Checked Then

' BenefitsCostDecimal += Decimal.Round(RETIREMENT\_STANDARD\_DECIMAL \* GrossPayDecimal, 2)

'Else

' 'No charge for not taking retirement benefit

'End If

'Use the retirement rate set in the CheckedChanged event

'for the retirement radio button controls

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

'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("C")

FederalTaxTextBox.Text = FederalTaxDecimal.ToString("N")

BenefitsTextBox.Text = BenefitsCostDecimal.ToString("N")

NetPayTextBox.Text = NetPayDecimal.ToString("C")

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

'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

Me.Close()

End If

End Sub

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

'Declare constant used in this sub procedure

Const MAX\_HOURS\_DECIMAL 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) > MAX\_HOURS\_DECIMAL) Then

'Hours must be numeric and within allowable range

e.Cancel = True

ErrorProvider1.SetError(HoursTextBox, "Hours worked must be a number between 0 and " & MAX\_HOURS\_DECIMAL.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

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

'Set focus to name textbox

NameTextBox.Focus()

End Sub

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

'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 = RETIREMENT\_STANDARD\_DECIMAL

Case "Retirement401ARadioButton" '401A rate

RetirementRateDecimal = RETIREMENT\_401A\_DECIMAL

End Select

End Sub

End Class