

Programming Logic - Intermediate

152-102

Input Validation

Notes	Activity
-------	----------

Text References

- [Input Validation](#) Page 238 – 240
- [Business Class](#) no book reference
- [Selecting the Proper Control](#) no book reference
- [Key-Level Validation](#) no book reference
- [MaxLength Property](#) Page 701
- [Min-Max Date Property](#)
- [MaskedTextBox \(Details\)](#) no book reference
- [Field-Level Validation](#) Page 311 – 316
- [GUI Validation](#) no book reference
- [Validating Event](#) Pages 312 – 316
- [Business Class Preparation](#) Pages
- [Business Class Validation](#) Page 748 – 750
- [Exception Handling](#) Pages 146 - 150
- [Error Provider Tool](#) no book reference
- [Business Class Load Combo Box](#)
- [Form-Level Validation](#) no book reference

- [Programmer defined procedures](#) Pages 352 – 362,
370 – 377
- [Calling a procedure](#) Pages 354, 371-2
- [Create Form-Level Validation Function](#)
- [Adding Class-Level Validation](#)

- [IsNumeric, isDate](#) Pages 221,
225 – 227

- [Appendix A: MaskedTextBox Details](#)
- [Appendix B: Common Validation Examples](#)

Notes	Activity
-------	----------

Work with Pizza Order
from Unit 1

Input Validation

Input validation is one of the most important and most often used programming techniques you'll learn. Almost every application includes some kind of input validation.

Input validation is the process of inspecting input values and determining whether they are valid. The accuracy of your program's output is only as good as the accuracy of its inputs. It is important that your application perform input validation on the values entered (or not entered) by the user.

Input validation can be done at three different times:

- Immediately after each key the user presses (key-level)
- Immediately after the user enters a value (individual values) (field-level)
- Just before calculations/processing are done (all values as a group) (processing/form level)

When you do validation depends on your preferences and often depends on the application itself.

This unit will focus on identifying errors, notifying the user of the error and allowing them to change the error before any processing (calculations, saving) is completed.

The Business Class

- Most (all?) validation is accomplished by a program's *business class (or classes)*.
- The business class contains all the business logic that must be incorporated into a program
 - The business logic includes rules for what defines valid data and what defines a valid record.
- For now, there is only one other kind of class: the GUI (graphical user interface) class.
 - In VB, this class is implemented by a form
- Later, we'll also incorporate a *data class* to complete our [*3-tier program model*](#).

Review the business class
for a Pizza Order

Notes	Activity
-------	----------

Selecting the Proper Control

Review the members and selected control

- The first step in ensuring the user provides valid input is to use the most appropriate controls to gather their inputs.
 - The controls appear in the GUI class.
- Many controls have built-in features that limit the type and range of data they will accept. These features contribute to valid inputs.
- Use checkboxes to get Yes/No type input from the user.
 - Yes / No radio buttons can also be effective.
- Use DateTimePickers to get dates from users
 - Set the MinDate and/or MaxDate properties to restrict the range of dates the user can enter.
 - A MonthView can also be used, but it takes up more screen space
- Use radio buttons when the list of choices is small.
- Use a ListBox to provide the user a fixed list of choices.
 - List is scrollable; can be quite long.
 - Use in place of lots of radio buttons
- Use a ComboBox to provide the user a list of choices and potentially the option to enter data that is not on the list
 - Uses less screen space than a list box
 - Use the Auto Complete feature to help the user quickly make selections (with fewer errors).
- Use a NumericUpDown to get numbers from the user.
 - Works best for whole numbers
 - Users can click arrows to change the value or simply type a value.
 - Use the Minimum and Maximum properties to limit the range of acceptable values.
- TextBoxes are the most frequently used input control; however, you should use the other controls whenever possible.

Validation Levels

- As I stated before, input validation can occur at three times: after each key is pressed, immediately after the user enters a value or before any calculations are done.
- I consider these three different *levels* of validation:
 - Key-level validation
 - Field-level validation
 - Form-level (or Class-level) validation

Notes

Activity

Key-Level Validation

- Key-level validation occurs as the user is pressing keys.
- Key-level validation is defined in the GUI class
- Key-level validation allows you to restrict what characters a field will accept. Some examples are:
 - Only digits for zip codes
 - Only letters for state codes
 - Only digits, commas, decimal points, etc. for numbers
- The KeyPress event gives us the access we need
 - The sender parameter tells us which object the user is currently typing in.
 - The e parameter (Event parameter), contains a KeyChar property that we can use to figure out which key was pressed.
 - Unfortunately, the KeyChar property is a character which the VB Select statement doesn't handle very well. The first thing we have to do is convert the KeyChar property to a string (which Select does handle).
 - A character can be promoted to a string, so all we have to do is assign the KeyChar to a string variable.
 - Now, in a Select statement, we'll list (using literals), the keys we want to **accept**.
 - We'll use a Select for readability—it's easier to add keys than it would be in an If statement.

Write txtCheeseSticks
KeyPressed event

Select Case thekey
Case "0" to "9"

Notes	Activity
<ul style="list-style-type: none"> – We'll also want to include the backspace key so the user can correct typing errors. <ul style="list-style-type: none"> ▪ Since we can't type a backspace between two quotes, we'll use the <code>ControlChars enumeration</code> ▪ Note the editing keys (arrows, del, home, end) are not captured by <code>KeyPress</code> (they are always accepted). If you need to capture these, use the <code>KeyDown</code> event. – If you want the user to be able to enter signs (- +), commas, decimal points, dollar signs, percent symbols, etc. in their numbers, you'll have to include those string literals in the case statement as well. <ul style="list-style-type: none"> ▪ Note: all the <i>acceptable</i> literals are included in one case option, separated by commas ▪ If you don't include a minus sign, you don't have to worry about negative numbers in field testing. 	<p>(Form-level so all procedures can use it) <code>Const BACKSP as String = ControlChars.Back</code></p> <p>Case "0" to "9", _ BACKSP, ",", (comma)</p>
<ul style="list-style-type: none"> ➤ OK, now we've told the module which keys to accept. But what code do we enter in <code>KeyPress</code> to designate the was key accepted? <ul style="list-style-type: none"> – The <code>e</code> parameter also includes a <code>Handled</code> (Boolean) property that designates whether the <code>KeyPress</code> event <i>handled</i> the key that was pressed. – Set <code>e.Handled</code> to <code>False</code> if you want <code>KeyPress</code> to pass this key on (have the form <i>handle</i> it, key is acceptable for the form to handle). – Set <code>e.Handled</code> to <code>True</code> if you want <code>KeyPress</code> to handle this key. If <code>KeyPress</code> handles it, the form effectively ignores the key. 	<p>Case "0" to "9", _ BACKSP <code>e.Handled = False</code></p> <p>Case Else <code>e.Handled = True</code></p> <p>End Select</p>
<ul style="list-style-type: none"> ➤ See the appendix to these notes for sample <code>KeyPressed</code> events ➤ More complete <code>KeyPressed</code> events for currency values, percentages and numbers with decimal places are available for your use in <code>VolkerTools</code> 	<p>Add <code>txtNumPies</code> to <code>handles</code> clause</p> <p>Review <code>KeyPressed</code> events in <code>VolkerTools</code> Copy Decimal into project</p>

Notes

Activity

MaxLength Property

- Another simple way to restrict what the user can enter is to change a text box's MaxLength property.
 - This restricts the number of characters a user can type in a text box
 - E.g. Set the MaxLength property to 2 for state codes
 - E.g. Set the MaxLength for a Quantity field to 3 if no more than 225 items can be ordered.
 - E.g. Set the MaxLength for Zip codes to 5.
- Max length restrictions should be defined in the business class and then referenced by the GUI class
 - Define public constants in the business class for each max length
 - When the GUI class loads (Form Load) reference the max lengths and set the max length property of the appropriate fields.
- Set max lengths for numeric fields with caution. These types of fields are very likely to change lengths.

Define MaxLength in business class
CustomerFirst to 15
CustomerLast to 20

```
Private Sub frmJellyBean_Load(ByVal sender As System.Object, _
                             ByVal e As System.EventArgs) Handles MyBase.Load
    txtCustomerFirst.MaxLength = JellyBeanOrder.MaxFirstNameLength
End Sub
```

Max and Min Date Properties

- Another simple technique you can use to help ensure valid data is to limit the dates a user can select from a date picker.
 - Date pickers include two properties that allow you to specify the first (minimum) and last (maximum) date the date picker will accept. (MinDate, MaxDate)
 - The date picker will not allow the user to select dates that are out of range
 - When running under Vista or Windows 7, the date pickers don't even display dates that are out of range.
- Define a minimum and maximum date value in the business class

Define min and max order date in business class
(5/15/1975, Today)

```
Public Const minOrderDate as Date = #6/1/2000#
Public Const maxOrderDate as Date = #6/1/2008#
```

Notes	Activity
-------	----------

- Often, you will want to use the current date as the maximum value (or some date related to the current date)
- Unfortunately, VB will not allow you to assign *Today* to a constant.
- This is the work around


```
Public Shared ReadOnly maxOrderDate As Date = Today
```

 - Public makes maxOrderDate available to the GUI
 - Shared ensures there's only one copy is created (shared) for all instances.
 - Public Const are shared by default. That's why we didn't designate *shared* for the constants
 - ReadOnly keeps the user classes from changing this value essentially making it a constant.

- In Form Load, set the date picker's min and max date properties to the business class constants

Set the date picker min and max values in Form Load. Note problems with max (remove)

```
Private Sub frmJellyBean_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load

    txtCustomerFirst.MaxLength = JellyBeanOrder.MaxFirstNameLength

    dtpOrderDate.MinDate = JellyBeanOrder.minOrderDate
    dtpOrderDate.MaxDate = JellyBeanOrder.maxOrderDate

End Sub
```

Notes	Activity
Masked Textbox	For information only

- A masked textbox is a special type of textbox that allows you to designate at design time what characters the textbox will accept.
- The masked textbox uses the *Mask* property (similar to a template) to designate what characters the textbox will accept.
- Any character the program's user types that is not acceptable to the mask is ignored.
 - E.g. If the user presses letter keys when entering social security numbers, the mask ignores them
- Masked textboxes can also insert required characters automatically, such as the dashes in social security numbers, slashes in dates or the colon in a time.
- These features simplify data entry for the user, but can also provide a level of validation.
- Personal note. I find, except for predefined masks, using `KeyPressed` and custom code provides a more user-friendly validation than using masks.
- For more details on the Masked Textbox, see the additional information in the [appendix](#) below.

Field-Level Validation

- Field-level validation can be done as the user is leaving a field or as part of form-level validation (when the user thinks all data has been entered)
 - Which method you choose is a matter of personal preference. Some programmers like to let the user know they've entered a bad value right away. Others prefer to let the user complete data entry then check for all errors.
- This section describes how to check for an error immediately, as the user is leaving the field.
 - See Form-Level Validation below, for techniques that wait until the user has completed data entry.

Notes

Activity

Validating Event

- For our field-level validation, we will be taking advantage of the Validating event.
 - Validating occurs when focus leaves an object, but before it is set to the next object
- When the user leaves a data field, we will add code to immediately store the value they entered into the appropriate property in the business class.
- This will cause the *setter* code in the business class property to execute.

```
Private Sub txtCustomerFirst_Validating(ByVal sender As Object, _  
    ByVal e As System.ComponentModel.CancelEventArgs) _  
    Handles txtCustomerFirst.Validating  
  
    'This code will be expanded below.  
    currRecord.CustomerFirstName = txtCustomerFirst.Text  
  
End Sub
```

Business Class Preparation

- There is one circumstance where GUI to Business class communication is a challenge—when transferring blank data from a text box to a numeric field in the business class
 - Numeric fields can't handle empty strings
- To deal with this problem, we will define *null constants* for **every numeric field in the class**
 - The word *null* means *missing* in programming and database terminology.
 - We will define constants with ridiculous numeric values to represent a missing numeric value.

Define NULLSTICKS,
NULLCOUPON,
NULLPIES

Modify New to initialize to
null values

```
Public Const NULLPRICE As Double = Double.MinValue  
Public Const NULLQUANTITY As Integer = Integer.MinValue
```

Notes	Activity
-------	----------

- These are defined as public constants which provide access of these values to the GUI
- Public constants are automatically *shared* – only one copy of the constant is created for all instances of this type
 - Shared members are accessed using the business class name (not an instance name). See examples below.
- Integer and Double MinValue are defined as the smallest possible value the type can hold. This is our *ridiculous* value (price and quantity would never be this value).
- We will use these constants whenever we want to designate the price or length as *missing*

Business Class Validation

- Add appropriate If statements to the field *setter* event to ensure the data entered is correct
- If the data is not correct (error discovered), we need some way for the business class to communicate to the GUI class that the data the user entered is invalid.
- This communication is normally done using *exceptions*.
 - Exceptions are simply messages from one class to another.
 - They don't have to represent errors, but they often do.

Ensure the first name and last name fields are not blank

Exception Handling

- VB makes *exception handling* quite simple
- To send a message, you *throw an exception*, sending a message from the business class to the GUI class

Throw exception if they are

```
Throw New Exception("First name is a required field.")
```

- Note the required keywords
- Also note you designate the message of your choice in parenthesis after System.Exception

In GUI, catch the exception and display msgbox for now

Test

- When the business class throws an exception, the GUI class must *catch* and optionally process that exception.
- This is done in the same way we processed exceptions in Programming Logic – Beginning: using Try-Catch

Notes	Activity
-------	----------

```
Private Sub txtCustomerFirst_Validating(ByVal sender As Object, _
    ByVal e As System.ComponentModel.CancelEventArgs) _
    Handles txtCustomerFirst.Validating

    Try
        currRecord.CustomerFirstName = txtCustomerFirst.Text
        'Clear previous error messages here (if appropriate)
    Catch ex As Exception
        'Display error message here
    End Try

End Sub
```

- Validating Required Strings

```
If value<>"" Then
    _title = value
Else
    Throw New Exception("Title is a required field")
End If
```

- Validating Optional Numeric fields


```
If value=NULLPRICE OrElse _
    (value>=MinPrice AndAlso value<=MaxPrice) Then
    _price = value
Else
    Throw New Exception("Price must be between " & MinPrice & _
        " and " & MaxPrice & ".")
End If
```

- Validating Required Numeric fields

```
If value>=MinPrice AndAlso value<=MaxPrice Then
    _price = value
ElseIf value=NULLPRICE Then
    Throw New Exception("Price is a required field.")
Else
    Throw New Exception("Price must be between " & MinPrice & _
        " and " & MaxPrice & ".")
End If
```

Notes	Activity
<ul style="list-style-type: none"> • Instead of using the validating event when a check box changes, we'll use the CheckChanged event <ul style="list-style-type: none"> ➤ Later our programs will display statuses. Using CheckChanged instead of Validating updates the status sooner. ➤ Since there's no validation, you can probably combine all your checkbox validating events into one module. <ul style="list-style-type: none"> – I prefer to keep them separate incase other processing needs to be done 	<p>Add a CheckChanged event for all checkboxes.</p>

Using the Error Provider

- Some users (and therefore the programmers who support them) don't like getting one error message at a time when they make multiple errors. They'd rather see all their errors at once.
- VB includes a tool called an Error Provider.
 - This tool allows you to display a special *error symbol*  next to fields that contain an error.
 - Actually, the error symbol can be any icon you want. This symbol is the default.
 - You can display the symbol next to **all** fields with errors at the same time.
 - Displaying the symbol does not disrupt further processing like a Message Box would.
- First, add an Error Provider control to your project.
 - Since all fields share the Error Provider, it will show up in the *Component Tray* at the bottom of the screen
 - I change the name of the Error Provider to *errProv* (short, but descriptive), but you can name the provider whatever you want.
- The next thing I do is change the Blink Style property of the Error Provider to *Never Blink*.
 - The blinking is annoying (though it does catch the user's attention). If there are multiple errors, they blink in sequence instead of all at once.
 - Experiment with the three blink styles to determine which works best for you.
- Once you've added the Error Provider, you can set an error message for any (or all) objects on the form.

Add errProv to the form.

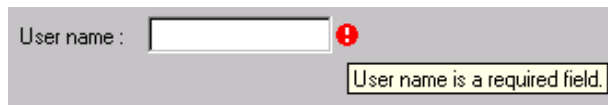
Convert txtCustomerFirst and txtCustomerLast _Validating to display/clear error marker

Notes

Activity

```
errProv.SetError(objname, "error message")
```

- Replace *objname* with the name of the object that contains the error.
- Replace *error message* with the appropriate error message to let the user know what they did wrong.
- When the user makes an error, the error icon will appear next to the object on the form.
- When the user touches the icon with their mouse, a tool tip appears displaying the error message you designated.
- A sample appears below showing the icon and error message. The mouse pointer is not shown but would be pointing to the error icon.



- Once an error icon is shown, it stays on the screen until your code gets rid of it. The way to get rid of it is to set the error message for this object to the empty string.
- Sample code:

```
Private Sub txtCustomerFirst_Validating(ByVal sender As Object, _  
    ByVal e As System.ComponentModel.CancelEventArgs) _  
    Handles txtCustomerFirst.Validating  
  
    errProv.SetError(txtCustomerFirst, "") 'Clear error marker  
    Try  
        currRecord.CustomerFirstName = txtCustomerFirst.Text  
  
    Catch ex As Exception  
        errProv.SetError(txtCustomerFirst, ex.Message)  
  
    End Try  
  
End Sub
```

Notes	Activity
-------	----------

```

Private Sub txtPrice_Validating(ByVal sender As Object, _
    ByVal e As System.ComponentModel.CancelEventArgs) _
    Handles txtPrice.Validating

    errProv.SetError(txtPrice, "") 'Clear error marker
    Try
        If txtPrice <> "" Then 'User entered a value
            currRecord.Price = CDb1(txtPrice.Text)
        Else
            currRecord.Price = JellyBeanOrder.NULLPRICE
        End If

    Catch ex As Exception
        errProv.SetError(txtPrice, ex.Message)

    End Try

End Sub

```

- These subroutines first clear the error marker, assuming the new value is OK
- These subroutines then *try* to transfer what the user has entered in the text box to the corresponding property in the business class.
- If the business class **does** throw an exception, the Catch block is executed.
 - ex represents the exception that was thrown
 - SetError places an error icon on the appropriate field
 - ex.Message contains the error message designated by the business class when the exception was thrown
- Note, the Price validation doesn't have to check if the user has entered a valid number. The Key Pressed routines ensure the user enters numbers properly.

Validate:
 txtFirst required
 txtLast required
 txtPies >1 < 15 required
 txtSticks >0 < 100, optional
 txtCoupon > .50 <5
 optional
 dtpOrderDate (between
 min and max)
 All others (no validation,
 just transfer)

Format data in all fields if
 it's valid (vProperCase,
 currency)

Move all validation code to
 a Region

Notes	Activity
-------	----------

- Setting the error icon location
 - If you'd rather have the error icon appear on the left side of the field, or if the field is more than one line deep and you don't want the icon to appear (vertically) centered, you can change the location of the error icon.
 - You can set the location of the error icon differently for every object on the form.

```
errErrors.SetIconAlignment(objname, location)
```

- After you type the comma after *objname*, a list of available *locations* will appear.

- Changing the error icon spacing
 - You can also change how close the icon appears to its object.

```
errErrors.SetIconPadding(objname, adjust)
```

- *adjust* how many pixels to *change* the icon spacing by
 - is measured in pixels and must be specified as an integer.
 - Can be a negative number (moves icon closer)

Set the icon padding for txtCustomerFirst to -10 so icon doesn't appear in last name box

Form-Level Validation

- Some programmers (maybe reflecting user preference) prefer to allow the user to enter all data into a form before checking any fields. After all the data has been entered, the program checks all the fields and reports errors.
 - Because the business class is checking all the fields (properties) of the class, this type of validation could be referred to as *class-level validation*
- Even if you do field-level validation, your program should still do form-level validation before doing any processing (calculations, saving)
- Form-level validation can check for errors field-level validation can't catch.
 - Form-level validation can do *consistency checks*—checking the values of multiple fields to ensure they make sense **in relation to each other**
 - Example: job code is valid for a department
- Form-level validation is normally done just before a program attempts to process the inputs (do calculations or save the record).

For reference only

Notes

Activity

Using the Business Class to Load List/Combo Boxes

- In many circumstances, the list of values to be included in a combo box is determined by tables in a database.
 - The techniques described here also apply to list boxes
- If the list of acceptable items is fairly small and fairly static (unchanging), you can define the valid list in the business class, then use that list to populate the combo box on the form.
- This can be done in at least two ways
 - Create a string that contains all the valid values
 - Create an *array* of strings where each is one of the valid values
- Because the array technique is more robust (provides more accurate validation) and to introduce you to arrays, we'll use that technique.
- Arrays are pointers to a list of items that are all of the same type
 - Very similar to collections
 - All items are referred to by the same name (name of the array)
 - Each item has an index (starting at 0)
- For our introduction to arrays we will define public, read only (constant) arrays as part of the business class
 - The nice thing about arrays is they can be filled when they are defined

```
Public Shared ReadOnly ArrayName() As String = _
    {"value1", "value2", "value3", etc.}
```

- Replace ArrayName with an appropriate name
- () after the ArrayName are required—they're what designate this variable as an array
- Arrays can be of any type (including class types). This is an array of strings
- To immediately initialize the items of an array, add an equal sign after the type and then the list of values, inside { curly brackets }, separated by commas

Create an array of valid sodas

Cola
Diet Cola
Lemon Lime
Lemonade
Orange Spunk
Root Beer
Ginger Ale
Grape Surprise
Dew Caffeine

Create an array of valid sizes

Personal
Small
Medium
Large

Notes	Activity
<ul style="list-style-type: none"> The form can now request the list of array values and use them to populate the combo or list box. <pre> For Each itm as String in ClassName.ArrayName 'Add itm to combo box Next </pre> <ul style="list-style-type: none"> ➤ Replace <code>ClassName</code> with the class name that contains the array ➤ Replace <code>ArrayName</code> with the name of the constant array <ul style="list-style-type: none"> Actually, a combo box is designed to accept an entire array of values the <code>AddRange</code> command, making this process even simpler. <pre> cmbBox.Items.AddRange(ClassName.ArrayName) </pre>	<p>Modify the form to load the flavors on form load (remove sodas from Items collection)</p>
<ul style="list-style-type: none"> The array can also be used to simplify validation in the business class <pre> 'In a setter If Array.IndexOf(_arrayName, value) <> -1 Then 'value is OK Else Throw error End if </pre> <ul style="list-style-type: none"> ➤ This code for the index of value in the array. If it is not there, <code>IndexOf</code> returns -1 ➤ Now, you can change the validating event for combo boxes that we wrote last semester to match our typical validating events. ➤ Note, this technique does not allow the field to be empty <ul style="list-style-type: none"> – If an empty value should be allowed (Soda) you could add the empty string to the array, but this will also add an blank entry in the combo box when it loads and will also add a empty line to the error message we will be creating below. – Alternatively, add a check for empty to the property validation <pre> 'In a setter If value="" OrElse Array.IndexOf(_arrayName, value) <> -1 Then 'value is OK Else Throw error End if </pre>	<p>Add validation to the soda property to ensure the selected soda is in the list.</p> <p>Add validation to the size property to ensure the size is in the list.</p> <p>Update the Validating events for Size and Soda in the GUI</p> <p>Update Soda validation to allow a blank entry</p>

Notes	Activity
-------	----------

Creating Programmer Defined Procedures

- VB provides the ability for you to write your own procedures (code not attached to an event) to perform a specific task.
 - This allows you to
 - break complicated code into parts that are manageable (*modularize*). Like breaking a book into chapters.
 - place code that is used by more than one procedure into a common location. Reduces duplicate code.
 - write reusable code that you can copy and use into other projects.
- VB provides two kinds of procedures: subroutines and functions
 - Subroutines do a process/perform a task, but don't return a value
 - Functions do a process/perform a task and return a value (one)
- Subroutines
 - General Format:

```
[Private | Public] Sub SubName (parm1 As type, parm2 As type, etc)
```

```
End Sub
```

Code

MakeStringFromArray
function to convert array
of strings to a single
string (or copy from
completed project)

Notes	Activity
-------	----------

- [Private | Public] (pick one)
 - Private for subroutines only used by one form
 - Public for subroutines **in a module** (used by more than one form)
- Sub – required. Indicates the start of the procedure. Let's VB know this is a subroutine, not a function.
- SubName – name you picked for this subroutine
 - Normally typed in *camel back* notation, no prefix, each word capitalized
 - Make the SubName meaningful
- parm1, parm2, etc – parameters for this subroutine
 - Optional. Subroutines don't have to have parameters
 - you pick the parameter name
 - Parameters are treated like local variables
- As *type* – type of each parameter.
 - Note parameters are separated with a comma.
- End Sub—normally entered automatically by VB. If not, be sure to add it to the end of your subroutine.

- Functions

- General Format:

[Private | Public] Function *FunctionName* (*parm1* As *type*, *parm2* As *type*, etc) As *functiontype*

Return *value*

End Function

Notes	Activity
<ul style="list-style-type: none">➤ [Private Public] (pick one)<ul style="list-style-type: none">– Private for functions only used by one form– Public for functions in a module (used by more than one form)➤ Function – required. Indicates the start of the procedure. Let's VB know this is a function, not a subroutine.➤ FunctionName – name you picked for this function<ul style="list-style-type: none">– Normally typed in <i>camel back</i> notation, no prefix– Make the FunctionName <u>meaningful</u>➤ parm1, parm2, etc – parameters for this function<ul style="list-style-type: none">– Optional. Functions don't have to have parameters– you pick the name– Parameters are treated like local variables➤ As <i>type</i> – type of each parameter.<ul style="list-style-type: none">– Note parameters are separated with a comma.➤ As <i>functiontype</i>—type of value this function returns<ul style="list-style-type: none">– Required. If the function doesn't return a value, use a subroutine➤ Return <i>value</i>—replace <i>value</i> with the variable that contains the value you want to return.<ul style="list-style-type: none">– Older versions of Visual Basic returned the value using the fuction name FunctionName = <i>somevalue</i> – This technique, though less self-documenting, is still available in VB.Net– VB is the only language I know of that provides this feature <ul style="list-style-type: none">• End Function—normally entered automatically by VB. If not, be sure to add it to the end of your function.	

Notes	Activity
-------	----------

Where to code User-defined Subs/Functions

- You can place form-level procedures anywhere inside the form's code **except**:
 - Must be below the *Class* statement for the form or module.
 - Cannot be inside another procedure
 - Must be **before** the End Class statement
 - As my projects get large, I alphabetize my procedures so they're easier to locate.
 - Global procedures (used by more than one form) must be stored in a module.
 - A module that holds Public procedures is just like the Documentation module we created in past units, but it holds code instead of comments.
 - Be sure to declare the procedure Public otherwise the forms won't be able access it.

Add Validating event for cmbSoda.

Ensure rdoCheckChanged transfers to business class.
Add a Try-Catch block

Creating a Form Validation Function

- Before the program attempts to process the form data (calc, save), it should ensure there are no error markers remaining on the form.
 - If errors remain, the processing is likely to fail or produce invalid results.
- We will use a function to do form-level validation. This function returns a Boolean value (T/F) designating whether any error markers remain on the form.
 - Since only this form will use this function, I define it in the form's code.

Create NoErrors function.

```
Private Function NoErrors( ) As Boolean
End Function
```

Notes	Activity
-------	----------

- This function will concatenate the Error Provider error messages of all the fields that could contain an error.
 - If there are no errors, this concatenation will be the empty string

```
Dim strErrors As String = ""
For Each ctrl As Control In Me.Controls
    strErrors &= errProv.GetError(ctrl)
Next
```

```
Return strErrors = ""
```

- The For Each loop steps through each control on the form, concatenating each object's error text to strErrors
 - Only the input objects could potentially have error text.
- Calling the validation function
 - To *call* (invoke) a subroutine or function, simply type the procedure's name at the point in your code where you want the call to occur.
 - If your subroutine/function requires parameters, you must provide values of the appropriate type for each parameter (in the parenthesis). Separate the parameters with commas.
 - If there are no parameters, you don't have to type the parenthesis—VB will add them automatically.
 - Normally, you'll call the validation function from your processing (Calc, Save) event.

Add the call to Save_Click

```
If NoErrors( ) Then
    'Do processing here
Else
    MessageBox.show(~~~~~)
End If
```

Notes	Activity
<ul style="list-style-type: none"> ➤ Because this is a call to a function, the function returns a value (Boolean value in this example). <ul style="list-style-type: none"> – That value can be immediately used in a statement (as above) or saved in a variable. – In this example, when NoErrors is called, the code in the function is executed. That code returns a value (true or false). That value is then immediately evaluated by the If statement to determine if its processing should be done. ➤ Note, there may not be an Else clause. If the entries are not valid, processing should not occur. I often add a Message Box to the Else clause to remind the user that errors exist. 	<p>Add the message box to the Else clause.</p>

Adding Class-Level Validation

- Just because there are no errors on the form doesn't mean the record is error free.
 - Some required fields may never have been visited
 - *Consistency errors* may still exist.
- Before saving or calculating within a class, the class should determine if all the fields are valid.
- If any errors remain, the business class must communicate that to the form
- My current technique is to create a function that checks every field in the class to ensure it is valid
 - To avoid duplicating the validation code of the class, I attempt (Try) to transfer EVERY data member into its corresponding property.
 - The only fields you don't need to include in InvalidFields are Boolean fields.
 - This causes the validation code within each property to execute.
 - Whenever an error is discovered, I concatenate the field name to a string of invalid field names
 - Once all the fields have been checked, I return the concatenated list of invalid fields to the Save module
 - If the list of fields is not blank, the Save module throws an exception.

Write InvalidFields function.

Write the SaveRecord method.
(if no invalid fields do nothing for now, else throw exception)

Modify btnSaveClick to Try to Save the record. If successful, display results, otherwise display exception.
(remove code that manually transfers field data to currRecord).

Notes

Activity

isNumeric and isDate

- Many other sample programs use isNumeric and isDate to help with validation
 - Our advanced KeyPressed events and Date Pickers have all but eliminated the need for these procedures (at least in validation)
- Remember that CInt, CDate and all their cousins are not very tolerant of strings sent to them with unexpected characters. These Boolean functions can check a string for acceptability before you send the string to a conversion function
- isNumeric checks to ensure a string contains a valid number.
 - isNumeric does not accept the empty string (returns False)
 - isNumeric will accept commas and dollar signs
 - isNumeric **does not** check to see if the commas are in appropriate positions. That's OK (sort of) because CInt (etc.) doesn't either
 - isNumeric **does** check to see if there's more than one decimal point in a number
 - The dollar sign can be at the beginning or end of the string only
 - Bottom line, if isNumeric returns True, CInt (etc.) can convert it.
- isDate will accept just about any kind of date
 - DatePickers have removed a lot of the need for this function
 - Dates can be entered using month names or numbers
 - Dates can include a year or not (use current year)
 - isDate does know how many days are in each month and recognizes leap years
- Example

```
If Not isNumeric(txtInput.Text) Then  
    MessageBox.Show(~~~~~)
```

Notes	Activity
-------	----------

Appendix A: Masked Textbox Details

- A masked textbox is a special type of textbox that allows you to designate at design time what characters the textbox will accept.
- The masked textbox uses the *Mask* property (similar to a template) to designate what characters the textbox will accept.
- Any character the program's user types that is not acceptable to the mask is ignored.
 - E.g. If the user presses letter keys when entering social security numbers, the mask ignores them
- Masked textboxes can also insert required characters automatically, such as the dashes in social security numbers, slashes in dates or the colon in a time.
- These features simplify data entry for the user, but can also provide a level of validation.
- The masked textbox comes with many predefined masks for common input data
 - Social Security Number (works very well)
 - Zip code (listed as Number) (works very well)
 - Phone Number (version without area code works very well)
 - Others such as Time, Phone with Area Code, and date formats are not as user-friendly

- Masked Textbox Properties
 - Text. Similar to the textbox Text property. The TextMaskFormat property determines whether the prompt characters and/or literals (automatically inserted characters) are included in the Text property.
 - PromptChar. Defines the character that is used to let the user know a character is expected—a placeholder. By default, the PromptChar is the underscore (_).
 - Often, the # is used to designate a digit is expected.
 - The masked textbox can only have one PromptChar
 - SkipLiterals. Normally set to True. Allows the user to enter the literal characters (automatically entered characters) without causing errors.
 - BeepOnError. When set to True, the speaker sounds when the user presses a key not allowed by the mask.
 - HidePromptOnLeave. When set to true, the mask characters are only shown when the masked textbox has focus (hide when the control does not have focus)
 - MaskCompleted. Available at runtime only. Designates whether all required characters in the mask have been filled in (Boolean)
 - MaskFull. Available at runtime only. Designates whether ALL characters (required and optional) have been entered.
 - Other commonly modified properties: Left, Top, Height, Width, Font, ForeColor, BackColor, Image, ImageAlign, TextAlign, Visible

- Masked textbox events
 - MaskInputRejected. Occurs when the user presses a character that is not acceptable by the mask. Allows you to display your own error messages when the user presses an invalid key.

Experiment with
PromptChar

BeepOnError

HidePromptOnLeave

TextMaskFormat (display
Text property of phone
number in label to see
results)

Maybe add a message box
to MaskInputRejected for
demonstration purposes
only

- Custom Masks
 - If the predefined masks do not meet your needs, you can create a mask of your own. The following characters are the most commonly used to defined masks.
 - 0 – required digit
 - 9 – optional digit
 - # - optional digit, space or + / -

 - L – required letter
 - ? – optional letter

 - < - convert letters to lowercase
 - > - convert letters to uppercase
 - Example. If your company uses a part number the consists of two capital letters followed by four digits, you could use this mask:
 - >LL0000
 - > automatically capitalizes the letters
 - LL forces the user to start the part number with 2 letters
 - 0000 forces the user to enter 4 digits after the letters
 - Search for *mask* in the help system for additional information and examples of custom masks.
- Personal note. I find, except for predefined masks described as working well above, using KeyPressed and custom code provides a more user-friendly validation than using masks.

Create a custom phone mask in European format:

999.000.0000

Enter sample data

Appendix B: Examples of Common Validation

```

`This function removes special characters from strings, leaving only digits.
` The second parameter designates the base of the number: Dec, Hex, Bin, or Oct
Public Function DigitsOnly(ByVal orgString As String, _
    Optional ByVal base As String = "Dec") As String
    Dim newString As String = ""           'orgString with non-digit characters removed
    Dim digits As String                  'digits to keep

    Select Case base
        Case "Hex" : digits = "0123456789ABCDEF"
        Case "Bin" : digits = "01"
        Case "Oct" : digits = "01234567"
        Case Else  : digits = "0123456789"
    End Select

    For c As Integer = 0 To orgString.Length - 1
        If digits.Contains(orgString.Chars(c)) Then
            newString &= orgString.Chars(c)
        End If
    Next

    Return newString
End Function

`This function validates an IP address.
Private Function IP_OK() As Boolean
    'This function assumes KeyPress ensures only digits and dots are entered
    'This function assumes MaxLength is set to 15 (###.###.###.###)

    'This function will have to be modified for IPv6.

    Dim oct() As String = {"", "", "", ""}           'Four octets
    Dim msg As String = ""                          'Error message to be displayed

    oct = txtIP.Text.Split(".")                    'c converts to character

    IP_OK = True 'Only return True when input is valid

    If txtIP.Text = "" Then 'Optional
        msg = "IP address is a required field."
        IP_OK = False
    ElseIf oct.Length <> 4 Then
        msg = "IP addresses must have four octets."
    Else
        For o As Integer = 0 To 3 'Check each octet for range
            If Cint(oct(o)) > 255 Then
                msg &= "Octet " & o + 1 & " is greater than 255. "
                IP_OK = False

                'Additional validation could be done here if additional
                'IP range restrictions exists (reserved IPs)
            End If
        Next
    End If

    errProv.SetError(txtIP, msg)

End Function

```

This function validates a MAC address.

```
Private Function MAC_OK() As Boolean
    'This function assumes KeyPress ensures only hex digits and dashes are entered
    'This function assumes MaxLength is set to 17 (HH-HH-HH-HH-HH-HH)
    'This function assumes character casing is UPPER.

    Dim mac As String = HexDigitsOnly(txtMAC.Text)    'Phone with dashes removed
    Dim msg As String = ""                            'Error message to be displayed

    MAC_OK = False 'Only return True when input is valid

    If mac = "" Then 'Optional
        msg = "MAC address is a required field."
    ElseIf mac.Length <> 12 Then
        msg = "MAC addresses must contain 12 hexadecimal digits."
    Else
        MAC_OK = True

        'Ensure user input is correct format (optional)
        'txtMAC.Text = CDb1(mac).ToString("HH-HH-HH-HH-HH-HH")
    End If

    errProv.SetError(txtMAC, msg)

End Function
```

This function validates a phone number (area code is optional).

```
Private Function PhoneOK() As Boolean
    'This function assumes KeyPress ensures only digits, spaces, parenthesis and dashes are entered
    'This function assumes MaxLength is set to 14 (999) 123-1234
    ' Adjust if your format is different

    Dim phone As String = DigitsOnly(txtPhone.Text)    'Phone with dashes removed
    Dim msg As String = ""                            'Error message to be displayed

    PhoneOK = False 'Only return True when input is valid

    If phone = "" Then 'Optional
        msg = "Phone number is a required field."
    ElseIf phone.Length <> 10 AndAlso phone.Length <> 7 Then
        msg = "Phone number must contain 7 or 10 digits."
    Else
        PhoneOK = True

        'Ensure user input is correct format (optional)
        If phone.Length = 7 Then
            txtPhone.Text = CDb1(phone).ToString("(715) 000-0000") 'Default area code is optional
        Else
            txtPhone.Text = CDb1(phone).ToString("(000) 000-0000") 'Adjust format if necessary
        End If
    End If

    errProv.SetError(txtPhone, msg)

End Function
```

'This function validates a social security number.

```
Private Function SSN_OK() As Boolean
    'This function assumes KeyPress ensures only digits and dashes are entered
    'This function assumes MaxLength is set to 11

    Dim ssn As String = DigitsOnly(txtSSN.Text)    'SSN with dashes removed
    Dim msg As String = ""                        'Error message to be displayed

    SSN_OK = False 'Only return True when input is valid

    If ssn = "" Then
        msg = "SSN is a required field."
    ElseIf ssn.Length <> 9 Then
        msg = "SSN must contain 9 digits."
    Else
        SSN_OK = True

        'Ensure user input is correct format (optional)
        txtSSN.Text = CDb1(ssn).ToString("000-00-0000")
    End If

    errProv.SetError(txtSSN, msg)

End Function
```

'This function validates a zip code include those with +4.

```
Private Function ZipOK() As Boolean
    'This function assumes KeyPress ensures only digits and dashes are entered
    'This function assumes MaxLength is set to 10 00000-9999
    'Adjust if your format is different

    Dim zip As String = DigitsOnly(txtZip.Text)    'Zip code with dashes removed
    Dim msg As String = ""                        'Error message to be displayed

    ZipOK = False 'Only return True when input is valid

    If zip = "" Then 'Optional
        msg = "Zip code is a required field."
    ElseIf zip.Length <> 9 AndAlso zip.Length <> 5 Then
        msg = "Zip code must contain 5 or 9 digits."
    Else
        ZipOK = True

        'Ensure user input is correct format (optional)
        If zip.Length = 5 Then
            txtZip.Text = CDb1(zip).ToString("00000") 'In case user inserted dash
        Else
            txtZip.Text = CDb1(zip).ToString("00000-0000") 'Adjust format if necessary
        End If
    End If

    errProv.SetError(txtZip, msg)

End Function
```