Programming Microsoft Windows with Visual Basic

2. The Visual Basic Language

Review and Preview

- Last week, we found there were three primary steps involved in developing an application using Visual Basic:

  1. Draw the user interface
  2. Assign properties to controls
  3. Attach code to events

  This week, we are primarily concerned with Step 3, attaching code. We will become more familiar with moving around in the Code window and learn some of the elements of the Basic language.

A Brief History of Basic

- Language developed in early 1960's at Dartmouth College:

  B (eginner's)
  A (All-Purpose)
  S (Symbolic)
  I (Instruction)
  C (Code)

- Answer to complicated programming languages (FORTRAN, Algol, Cobol ...). First timeshare language.

- In the mid-1970's, two college students write first Basic for a microcomputer (Altair) - cost $350 on cassette tape. You may have heard of them: Bill Gates and Paul Allen!

  - Every Basic since then essentially based on that early version. Examples include: GW-Basic, QBasic, QuickBasic.

- Visual Basic was introduced in 1991.

Visual Basic Statements and Expressions

- The simplest statement is the **assignment** statement. It consists of a variable name, followed by the assignment operator (=), followed by some sort of **expression**.

  Examples:

  ```
  StartTime = Now
  Explorer.Caption = "Captain Spaulding"
  BitCount = ByteCount * 8
  Energy = Mass * LIGHTSPEED ^ 2
  NetWorth = Assets - Liabilities
  ```

  The assignment statement stores information.

  - Statements normally take up a single line with no terminator. Statements can be **stacked** by using a colon (**:**) to separate them. Example:

    ```
    StartTime = Now : EndTime = StartTime + 10
    ```

  (Be careful stacking statements, especially with If/End If structures. You may not get the response you desire.)

  - If a statement is very long, it may be continued to the next line using the **continuation** character, an underscore (**_**). Example:

    ```
    Months = Log(Final * IntRate / Deposit + 1) _
    / Log(1 + IntRate)
    ```

- Comment statements begin with the keyword **Rem** or a single quote (**'**). For example:

  ```
  Rem This is a remark
  ' This is also a remark
  x = 2 * y ' another way to write a remark or comment
  ```

  You, as a programmer, should decide how much to comment your code. Consider such factors as reuse, your audience, and the legacy of your code.

Visual Basic Operators

- The simplest **operators** carry out **arithmetic** operations. These operators in their order of precedence are:

Operator	Operation
^	Exponentiation
* /	Multiplication and division
\	Integer division (truncates)
Mod	Modulus
+ -	Addition and subutraction

- **Parentheses** around expressions can change precedence.

- To **concatentate** two strings, use the **&** symbol or the **+** symbol:

```
lblTime.Caption = "The current time is" & Format(Now, “hh:mm”)
txtSample.Text = "Hook this “ + “to this”
```

- There are six **comparison** operators in Visual Basic:

Operator	Comparison
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal to

- The result of a comparison operation is a Boolean value (**True** or **False**).

- We will use three **logical** operators

Operator	Operation
Not	Logical not
And	Logical and
Or	Logical or

- The **Not** operator simply negates an operand.

- The **And** operator returns a True if both operands are True. Else, it returns a False.

- The **Or** operator returns a True if either of its operands is True, else it returns a False.

- Logical operators follow arithmetic operators in precedence.

Visual Basic Functions

- Visual Basic offers a rich assortment of built-in **functions**. The on-line help utility will give you information on any or all of these functions and their use. Some examples are:

Function	Value Returned
Abs	Absolute value of a number
Asc	ASCII or ANSI code of a character
Chr	Character corresponding to a given ASCII or ANSI code
Cos	Cosine of an angle
Date	Current date as a text string
Format	Date or number converted to a text string
Left	Selected left side of a text string
Len	Number of characters in a text string
Mid	Selected portion of a text string
Now	Current time and date
Right	Selected right end of a text string
Rnd	Random number
Sin	Sine of an angle
Sqr	Square root of a number
Str	Number converted to a text string
Time	Current time as a text string
Timer	Number of seconds elapsed since midnight
Val	Numeric value of a given text string

A Closer Look at the Rnd Function

- In writing games and learning software, we use the **Rnd** function to introduce randomness. This insures different results each time you try a program. The Visual Basic function Rnd returns a single precision, random number between 0 and 1 (actually greater than or equal to 0 and less than 1). To produce random integers (I) between Imin and Imax, use the formula:

```
I = Int((Imax - Imin + 1) * Rnd) + Imin
```

- The random number generator in Visual Basic must be seeded. A **Seed** value initializes the generator. The **Randomize** statement is used to do this:

```
Randomize Seed
```

If you use the same Seed each time you run your application, the same sequence of random numbers will be generated. To insure you get different numbers every time you use your application (preferred for games), use the **Timer** function to seed the generator:

```
Randomize Timer
```

Place this statement in the **Form_Load** event procedure.

- **Examples**:

To roll a six-sided die, the number of spots would be computed using:

```
NumberSpots = Int(6 * Rnd) + 1
```

To randomly choose a number between 100 and 200, use:

```
Number = Int(101 * Rnd) + 100
```

Example 2-1

Savings Account

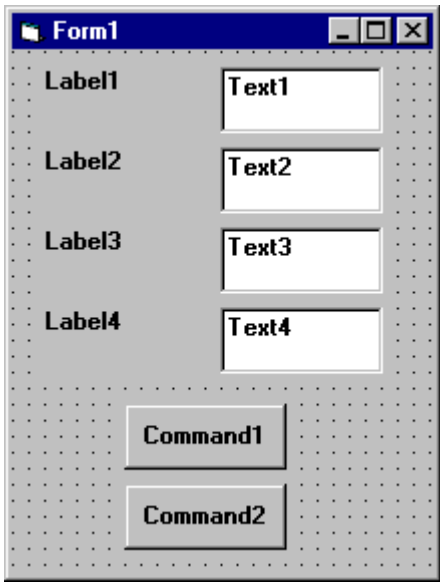
1. Start a new project. The idea of this project is to determine how much you save by making monthly deposits into a savings account. For those interested, the mathematical formula used is:

$$F = D [(1 + I)^M - 1] / I$$

where

F - Final amount
D - Monthly deposit amount
I - Monthly interest rate
M - Number of months

2. Place 4 label boxes, 4 text boxes, and 2 command buttons on the form. It should look something like this:



3. Set the properties of the form and each object.

Form1:

BorderStyle	1-Fixed Single
Caption	Savings Account
Name	frmSavings

Label1:

Caption	Monthly Deposit

Label2:

Caption	Yearly Interest

Label3:

Caption	Number of Months

Label4:

Caption	Final Balance

Text1:

Text	[Blank]
Name	txtDeposit

Text2:

Text	[Blank]
Name	txtInterest

Text3:

Text	[Blank]
Name	txtMonths

Text4:

Text	[Blank]
Name	txtFinal

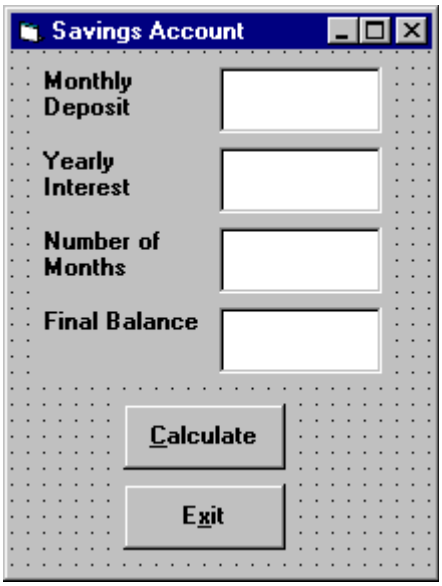
Command1:

Caption	&Calculate
Name	cmdCalculate

Command2:

Caption	E&xit
Name	cmdExit

Now, your form should look like this:



4. Declare four variables in the **general declarations** area of your form. This makes them available to all the form procedures:

```
Option Explicit
Dim Deposit As Single
Dim Interest As Single
Dim Months As Single
Dim Final As Single
```

The **Option Explicit** statement forces us to declare all variables.

5. Attach code to the **cmdCalculate** command button **Click** event.

```
Private Sub cmdCalculate_Click ()
Dim IntRate As Single
'Read values from text boxes
Deposit = Val(txtDeposit.Text)
Interest = Val(txtInterest.Text)
IntRate = Interest / 1200
Months = Val(txtMonths.Text)
'Compute final value and put in text box
Final = Deposit * ((1 + IntRate) ^ Months - 1) / IntRate
txtFinal.Text = Format(Final, "#####0.00")
End Sub
```

This code reads the three input values (monthly deposit, interest rate, number of months) from the text boxes, computes the final balance using the provided formula, and puts that result in a text box.

6. Attach code to the **cmdExit** command button **Click** event.

```
Private Sub cmdExit_Click ()
End
End Sub
```

7. Play with the program. Make sure it works properly. Save the project.

Visual Basic Symbolic Constants

- Many times in Visual Basic, functions and objects require data arguments that affect their operation and return values you want to read and interpret. These arguments and values are constant numerical data and difficult to interpret based on just the numerical value. To make these constants more understandable, Visual Basic assigns names to the most widely used values - these are called **symbolic constants**. Appendix I lists many of these constants.

- As an example, to set the background color of a form named **frmExample** to blue, we could type:

```
frmExample.BackColor = 0xFF0000
```

or, we could use the symbolic constant for the blue color (**vbBlue**):

```
frmExample.BackColor = vbBlue
```

- It is strongly suggested that the symbolic constants be used instead of the numeric values, when possible. You should agree that **vbBlue** means more than the value **0xFF0000** when selecting the background color in the above example. You do not need to do anything to define the symbolic constants - they are built into Visual Basic.

Defining Your Own Constants

- You can also define your own constants for use in Visual Basic. The format for defining a constant named **PI** with a value **3.14159** is:

```
Const PI = 3.14159
```

- **User-defined constants** should be written in all upper case letters to distinguish them from variables. The scope of constants is established the same way a variables' scope is. That is, if defined within a procedure, they are local to the procedure. If defined in the general declarations of a form, they are global to the form. To make constants global to an application, use the format:

```
Global Const PI = 3.14159
```

within the general declarations area of a module.

Visual Basic Branching - If Statements

- **Branching** statements are used to cause certain actions within a program if a certain condition is met.

  - The simplest is the **If/Then** statement:

    ```
    If Balance - Check < 0 Then Print "You are overdrawn"
    ```

  Here, if and only if Balance - Check is less than zero, the statement “You are overdrawn” is printed.

- You can also have **If/Then/End If** blocks to allow multiple statements:

  ```
  If Balance - Check < 0 Then
    Print "You are overdrawn"
    Print "Authorities have been notified"
  End If
  ```

  In this case, if Balance - Check is less than zero, two lines of information are printed.

- Or, **If/Then/Else/End If** blocks:

  ```
  If Balance - Check < 0 Then
    Print "You are overdrawn"
    Print "Authorities have been notified"
  Else
    Balance = Balance - Check
  End If
  ```

  Here, the same two lines are printed if you are overdrawn (Balance - Check < 0), but, if you are not overdrawn (**Else**), your new Balance is computed.

- Or, we can add the **ElseIf** statement:

```
If Balance - Check < 0 Then
  Print "You are overdrawn"
  Print "Authorities have been notified"
ElseIf Balance - Check = 0 Then
  Print "Whew! You barely made it"
  Balance = 0
Else
  Balance = Balance - Check
End If
```

  Now, one more condition is added. If your Balance equals the Check amount (**ElseIf** Balance - Check = 0), a different message appears.

- In using branching statements, make sure you consider all viable possibilities in the If/Else/End If structure. Also, be aware that each If and ElseIf in a block is tested sequentially. The first time an If test is met, the code associated with that condition is executed and the If block is exited. If a later condition is also True, it will never be considered.

Key Trapping

- Note in the previous example, there is nothing to prevent the user from typing in meaningless characters (for example, letters) into the text boxes expecting numerical data. Whenever getting input from a user, we want to limit the available keys they can press. The process of interecepting unacceptable keystrokes is **key trapping**.

- Key trapping is done in the **KeyPress** procedure of an object. Such a procedure has the form (for a text box named **txtText**):

```
Sub txtText_KeyPress (KeyAscii as Integer)
        .
        .
        .
End Sub
```

  What happens in this procedure is that every time a key is pressed in the corresponding text box, the ASCII code for the pressed key is passed to this procedure in the argument list (i.e. **KeyAscii**). If KeyAscii is an acceptable value, we would do nothing. However, if KeyAscii is not acceptable, we would set KeyAscii equal to zero and exit the procedure. Doing this has the same result of not pressing a key at all. ASCII values for all keys are available via the on-line help in Visual Basic. And some keys are also defined by symbolic

constants. Where possible, we will use symbolic constants; else, we will use the ASCII values.

- As an example, say we have a text box (named **txtExample**) and we only want to be able to enter upper case letters (ASCII codes 65 through 90, or, correspondingly, symbolic constants **vbKeyA** through **vbKeyZ**). The key press procedure would look like (the **Beep** causes an audible tone if an incorrect key is pressed):

```
Sub txtExample_KeyPress(KeyAscii as Integer)
  If KeyAscii >= vbKeyA And KeyAscii <= vbKeyZ Then
    Exit Sub
  Else
    KeyAscii = 0
    Beep
  End If
End Sub
```

- In key trapping, it's advisable to always allow the backspace key (ASCII code 8; symbolic constant **vbKeyBack**) to pass through the key press event. Else, you will not be able to edit the text box properly.

Example 2-2

Savings Account - Key Trapping

1. Note the acceptable ASCII codes are 48 through 57 (numbers), 46 (the decimal point), and 8 (the backspace key). In the code, we use symbolic constants for the numbers and backspace key. Such a constant does not exist for the decimal point, so we will define one with the following line in the **general declarations** area:

```
Const vbKeyDecPt = 46
```

2. Add the following code to the three procedures: **txtDeposit_KeyPress**, **txtInterest_KeyPress**, and **txtMonths_KeyPress**.

```
Private Sub txtDeposit_KeyPress (KeyAscii As Integer)
'Only allow number keys, decimal point, or backspace
If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or KeyAscii
= vbKeyDecPt Or KeyAscii = vbKeyBack Then
  Exit Sub
Else
 KeyAscii = 0
 Beep
End If
End Sub

Private Sub txtInterest_KeyPress (KeyAscii As Integer)
'Only allow number keys, decimal point, or backspace
If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or KeyAscii
= vbKeyDecPt Or KeyAscii = vbKeyBack Then
  Exit Sub
Else
 KeyAscii = 0
 Beep
End If
End Sub
```

```
Private Sub txtMonths_KeyPress (KeyAscii As Integer)
'Only allow number keys, decimal point, or backspace
If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or KeyAscii
= vbKeyDecPt Or KeyAscii = vbKeyBack Then
  Exit Sub
Else
 KeyAscii = 0
 Beep
End If
End Sub
```

(In the If statements above, note the word processor causes a line break where there really shouldn't be one. That is, there is no line break between the words **Or KeyAscii** and **= vbKeyDecPt**. One appears due to page margins. In all code in these notes, always look for such things.)

3. Rerun the application and test the key trapping performance.

Select Case - Another Way to Branch

- In addition to If/Then/Else type statements, the **Select Case** format can be used when there are multiple selection possibilities.

  - Say we've written this code using the **If** statement:

```
If Age = 5 Then
  Category = "Five Year Old"
ElseIf Age >= 13 and Age <= 19 Then
  Category = "Teenager"
ElseIf (Age >= 20 and Age <= 35) Or Age = 50 Or (Age >= 60 and Age <=
65) Then
  Category = "Special Adult"
ElseIf Age > 65 Then
  Category = "Senior Citizen"
Else
  Category = "Everyone Else"
End If
```

  The corresponding code with **Select Case** would be:

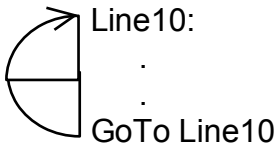
```
Select Case Age
  Case 5
    Category = "Five Year Old"
  Case 13 To 19
    Category = "Teenager"
  Case 20 To 35, 50, 60 To 65
    Category = "Special Adult"
  Case Is > 65
    Category = "Senior Citizen"
  Case Else
    Category = "Everyone Else"
End Select
```

  Notice there are several formats for the Case statement. Consult on-line help for discussions of these formats.

The GoTo Statement

- Another branching statement, and perhaps the most hated statement in programming, is the **GoTo** statement. However, we will need this to do Run-Time error trapping. The format is **GoTo *Label***, where ***Label*** is a labeled line. Labeled lines are formed by typing the *Label* followed by a colon.

  - **GoTo** Example:



  When the code reaches the GoTo statement, program control transfers to the line labeled Line10.

Visual Basic Looping

- Looping is done with the **Do/Loop** format. Loops are used for operations are to be repeated some number of times. The loop repeats until some specified condition at the beginning or end of the loop is met.

- **Do While/Loop** Example:

```
Counter = 1
Do While Counter <= 1000
  Debug.Print Counter
  Counter = Counter + 1
Loop
```

  This loop repeats as long as (**While**) the variable Counter is less than or equal to 1000. Note a Do While/Loop structure will not execute even once if the While condition is violated (False) the first time through. Also note the **Debug.Print** statement. What this does is print the value Counter in the Visual Basic Debug window. We'll learn more about this window later in the course.

- **Do Until/Loop** Example:

```
Counter = 1
Do Until Counter >  1000
  Debug.Print Counter
  Counter = Counter + 1
Loop
```

  This loop repeats **Until** the Counter variable exceeds 1000. Note a Do Until/Loop structure will not be entered if the Until condition is already True on the first encounter.

- **Do/Loop While** Example:

```
Sum = 1
Do
  Debug.Print Sum
  Sum = Sum + 3
Loop While Sum <= 50
```

  This loop repeats **While** the Variable Sum is less than or equal to 50. Note, since the While check is at the end of the loop, a Do/Loop While structure is always executed at least once.

- **Do/Loop Until** Example:

```
Sum = 1
Do
  Debug.Print Sum
  Sum = Sum + 3
Loop Until Sum > 50
```

  This loop repeats Until Sum is greater than 50. And, like the previous example, a Do/Loop Until structure always executes at least once.

- Make sure you can always get out of a loop! Infinite loops are never nice. If you get into one, try **Ctrl+Break**. That sometimes works - other times the only way out is rebooting your machine!

- The statement **Exit Do** will get you out of a loop and transfer program control to the statement following the Loop statement.

Visual Basic Counting

- Counting is accomplished using the **For/Next** loop.

  Example

```
For I = 1 to 50 Step 2
  A = I * 2
  Debug.Print A
Next I
```

  In this example, the variable I initializes at 1 and, with each iteration of the For/Next loop, is incremented by 2 (**Step**). This looping continues until I becomes greater than or equal to its final value (50). If Step is not included, the default value is 1. Negative values of Step are allowed.

- You may exit a For/Next loop using an **Exit For** statement. This will transfer program control to the statement following the **Next** statement.

Example 2-3

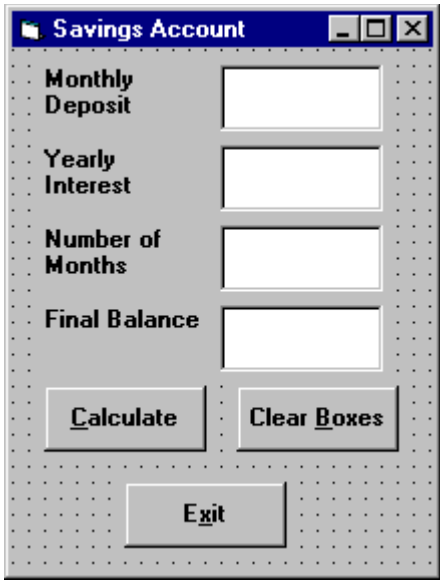
Savings Account - Decisions

1. Here, we modify the Savings Account project to allow entering any three values and computing the fourth. First, add a third command button that will clear all of the text boxes. Assign the following properties:

   Command3:

Caption	Clear &Boxes
Name	cmdClear

   The form should look something like this when you're done:



2. Code the **cmdClear** button **Click** event:

```
Private Sub cmdClear_Click ()
'Blank out the text boxes
txtDeposit.Text = ""
txtInterest.Text = ""
txtMonths.Text = ""
txtFinal.Text = ""
End Sub
```

   This code simply blanks out the four text boxes when the **Clear** button is clicked.

3. Code the **KeyPress** event for the **txtFinal** object:

```
Private Sub txtFinal_KeyPress (KeyAscii As Integer)
'Only allow number keys, decimal point, or backspace
If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or KeyAscii
= vbKeyDecPt Or KeyAscii = vbKeyBack Then
   Exit Sub
Else
  KeyAscii = 0
  Beep
End If
End Sub
```

We need this code because we can now enter information into the Final Value text box.

4. The modified code for the **Click** event of the **cmdCalculate** button is:

```
Private Sub cmdCalculate_Click()
Dim IntRate As Single
Dim IntNew As Single
Dim Fcn As Single, FcnD As Single
'Read the four text boxes
Deposit = Val(txtDeposit.Text)
Interest = Val(txtInterest.Text)
IntRate = Interest / 1200
Months = Val(txtMonths.Text)
Final = Val(txtFinal.Text)
'Determine which box is blank
'Compute that missing value and put in text box
If txtDeposit.Text = "" Then
'Deposit missing
  Deposit = Final / (((1 + IntRate) ^ Months - 1) /
IntRate)
  txtDeposit.Text = Format(Deposit, "#####0.00")
ElseIf txtInterest.Text = "" Then
'Interest missing - requires iterative solution
  IntNew = (Final / (0.5* Months * Deposit) - 1) /
Months
  Do
    IntRate = IntNew
    Fcn = (1 + IntRate) ^ Months - Final * IntRate /
Deposit - 1
    FcnD = Months * (1 + IntRate) ^ (Months - 1) -
Final / Deposit
    IntNew = IntRate - Fcn / FcnD
  Loop Until Abs(IntNew - IntRate) < 0.00001 / 12
```

```
  Interest = IntNew * 1200
  txtInterest.Text = Format(Interest, "##0.00")
ElseIf txtMonths.Text = "" Then
'Months missing
  Months = Log(Final * IntRate / Deposit + 1) / Log(1 +
IntRate)
  txtMonths.Text = Format(Months, "###.0")
ElseIf txtFinal.Text = "" Then
'Final value missing
  Final = Deposit * ((1 + IntRate) ^ Months - 1) /
IntRate
  txtFinal.Text = Format(Final, "#####0.00")
End If
End Sub
```

In this code. we first read the text information from all four text boxes and based on which one is blank, compute the missing information and display it in the corresponding text box. Solving for missing **Deposit**, **Months**, or **Final** information is a straightforward manipulation of the equation given in Example 2-2.

If the **Interest** value is missing, we have to solve an Mth-order polynomial using something called Newton-Raphson iteration - a good example of using a Do loop. Finding the **Interest** value is straightforward. What we do is guess at what the interest is, compute a better guess (using Newton-Raphson iteration), and repeat the process (loop) until the old guess and the new guess are close to each other. You can see each step in the code.

5. Test and save your application. Go home and relax.

Exercise 2-1

Computing a Mean and Standard Deviation

Develop an application that allows the user to input a sequence of numbers. When done inputting the numbers, the program should compute the mean of that sequence and the standard deviation. If N numbers are input, with the ith number represented by x_i, the formula for the mean ($\bar{x}$) is:

$$\bar{x} = (\sum_{i=1}^{N} x_i)/N$$

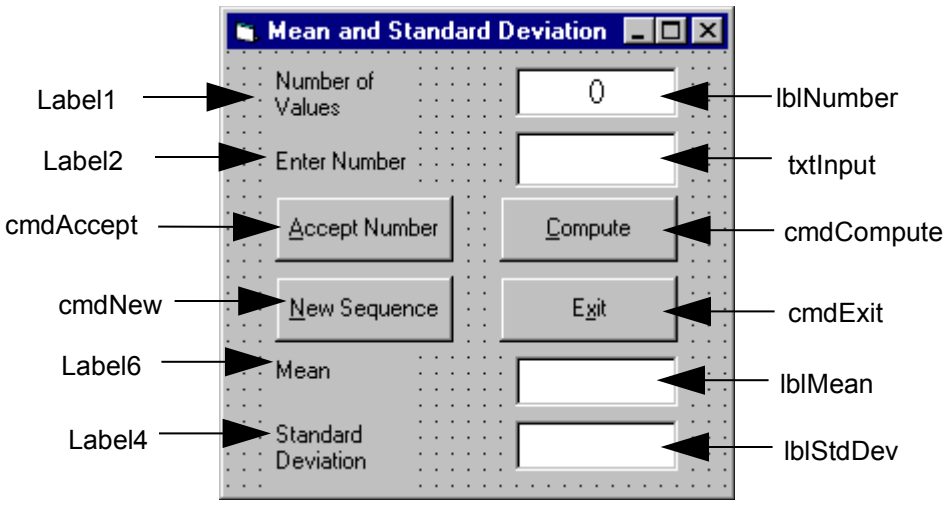
and to compute the standard deviation (s), take the square root of this equation:

$$s^2 = [N\sum_{i=1}^{N} x_i^2 - (\sum_{i=1}^{N} x_i)^2]/[N(N-1)]$$

The Greek sigmas in the above equations simply indicate that you add up all the corresponding elements next to the sigma.

My Solution:

Form:



Properties:

Form **frmStats**:
    Caption = Mean and Standard Deviation

CommandButton **cmdExit**:
    Caption = E&xit

CommandButton **cmdAccept**:
    Caption = &Accept Number

CommandButton **cmdCompute**:
    Caption = &Compute

CommandButton **cmdNew**:
    Caption = &New Sequence

TextBox **txtInput**:
    FontName = MS Sans Serif
    FontSize = 12

Label **lblStdDev**:
    Alignment = 2 - Center
    BackColor = &H00FFFFFF& (White)
    BorderStyle = 1 - Fixed Single
    FontName = MS Sans Serif
    FontSize = 12

Label **Label6**:
    Caption = Standard Deviation

Label **lblMean**:
    Alignment = 2 - Center
    BackColor = &H00FFFFFF& (White)
    BorderStyle = 1 - Fixed Single
    FontName = MS Sans Serif
    FontSize = 12

Label **Label4**:
    Caption = Mean

Label **lblNumber**:
Alignment = 2 - Center
BackColor = &H00FFFFFF& (White)
BorderStyle = 1 - Fixed Single
FontName = MS Sans Serif
FontSize = 12

Label **Label2**:
Caption = Enter Number

Label **Label1**:
Caption = Number of Values

Code:

General Declarations:

```
Option Explicit
Dim NumValues As Integer
Dim SumX As Single
Dim SumX2 As Single
Const vbKeyMinus = 45
Const vbKeyDecPt = 46
```

cmdAccept Click Event:

```
Private Sub cmdAccept_Click()
Dim Value As Single
txtInput.SetFocus
NumValues = NumValues + 1
lblNumber.Caption = Str(NumValues)
'Get number and sum number and number-squared
Value = Val(txtInput.Text)
SumX = SumX + Value
SumX2 = SumX2 + Value ^ 2
txtInput.Text = ""
End Sub
```

cmdCompute Click Event:

```
Private Sub cmdCompute_Click()
Dim Mean As Single
Dim StdDev As Single
txtInput.SetFocus
'Make sure there are at least two values
If NumValues < 2 Then
  Beep
  Exit Sub
End If
'Compute mean
Mean = SumX / NumValues
lblMean.Caption = Str(Mean)
'Compute standard deviation
StdDev = Sqr((NumValues * SumX2 - SumX ^ 2) / (NumValues *
(NumValues - 1)))
lblStdDev.Caption = Str(StdDev)
End Sub
```

cmdExit Click Event:

```
Private Sub cmdExit_Click()
End
End Sub
```

cmdNew Click Event:

```
Private Sub cmdNew_Click()
'Initialize all variables
txtInput.SetFocus
NumValues = 0
lblNumber.Caption = "0"
txtInput.Text = ""
lblMean.Caption = ""
lblStdDev.Caption = ""
SumX = 0
SumX2 = 0
End Sub
```

txtInput KeyPress Event:

```
Private Sub txtInput_KeyPress(KeyAscii As Integer)
'Only allow numbers, minus sign, decimal point, backspace,
return keys
If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or KeyAscii =
vbKeyMinus Or KeyAscii = vbKeyDecPt Or KeyAscii = vbKeyBack
Then
  Exit Sub
ElseIf KeyAscii = vbKeyReturn Then
  Call cmdAccept_Click
Else
  KeyAscii = 0
End If
End Sub
```

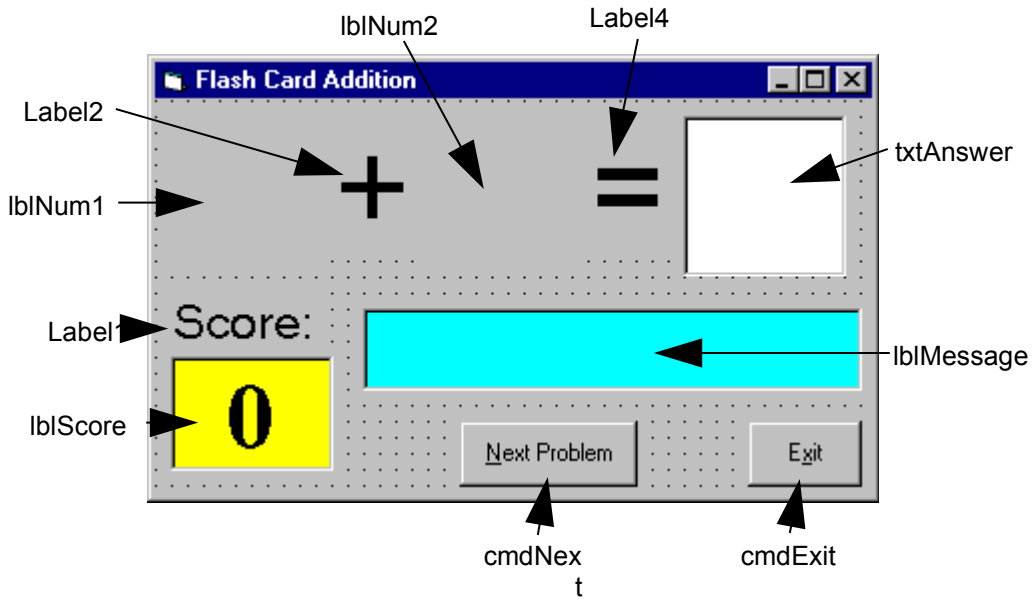
Exercise 2-2

Flash Card Addition Problems

Write an application that generates random addition problems. Provide some kind of feedback and scoring system as the problems are answered.

My Solution:

Form:



Properties:

Form **frmAdd**:
BorderStyle = 1 - Fixed Single
Caption = Flash Card Addition

CommandButton **cmdNext**:
Caption = &Next Problem
Enabled = False

CommandButton **cmdExit**:
Caption = E&xit

TextBox **txtAnswer**:
FontName = Arial
FontSize = 48
MaxLength = 2

Label **lblMessage**:
Alignment = 2 - Center
BackColor = &H00FFFF00& (Cyan)
BorderStyle = 1 - Fixed Single
FontName = MS Sans Serif
FontBold = True
FontSize = 24
FontItalic = True

Label **lblScore**:
Alignment = 2 - Center
BackColor = &H0000FFFF& (Yellow)
BorderStyle = 1 - Fixed Single
Caption = 0
FontName = Times New Roman
FontBold = True
FontSize = 36

Label **Label1**:
Alignment = 2 - Center
Caption = Score:
FontName = MS Sans Serif
FontSize = 18

Label **Label4**:
Alignment = 2 - Center
Caption = =
FontName = Arial
FontSize = 48

Label **lblNum2**:
Alignment = 2 - Center
FontName = Arial
FontSize = 48

Label **Label2**:
Alignment = 2 - Center
Caption = +
FontName = Arial
FontSize = 48

Label **lblNum1**:
Alignment = 2 - Center
FontName = Arial
FontSize = 48

Code:

General Declarations:

```
Option Explicit
Dim Sum As Integer
Dim NumProb As Integer, NumRight As Integer
```

cmdExit Click Event:

```
Private Sub cmdExit_Click()
End
End Sub
```

cmdNext Click Event:

```
Private Sub cmdNext_Click()
'Generate next addition problem
Dim Number1 As Integer
Dim Number2 As Integer
txtAnswer.Text = ""
lblMessage.Caption = ""
NumProb = NumProb + 1
'Generate random numbers for addends
Number1 = Int(Rnd * 21)
Number2 = Int(Rnd * 21)
lblNum1.Caption = Format(Number1, "#0")
lblNum2.Caption = Format(Number2, "#0")
'Find sum
Sum = Number1 + Number2
cmdNext.Enabled = False
txtAnswer.SetFocus
End Sub
```

Form Activate Event:

```
Private Sub Form_Activate()
Call cmdNext_Click
End Sub
```

Form Load Event:

```
Private Sub Form_Load()
Randomize Timer
NumProb = 0
NumRight = 0
End Sub
```

txtAnswer KeyPress Event:

```
Private Sub txtAnswer_KeyPress(KeyAscii As Integer)
Dim Ans As Integer
'Check for number only input and for return key
If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or KeyAscii =
vbKeyBack Then
  Exit Sub
ElseIf KeyAscii = vbKeyReturn Then
'Check answer
  Ans = Val(txtAnswer.Text)
  If Ans = Sum Then
    NumRight = NumRight + 1
    lblMessage.Caption = "That's correct!"
  Else
    lblMessage.Caption = "Answer is " + Format(Sum, "#0")
  End If
  lblScore.Caption = Format(100 * NumRight / NumProb, "##0")
  cmdNext.Enabled = True
  cmdNext.SetFocus
Else
  KeyAscii = 0
End If
End Sub
```

This page intentionally not left blank.