Chapter 5

Lists and Loops
• This chapter introduces:
  – Input boxes
  – List and combo boxes
  – Loops
  – Random numbers
  – The ToolTip control
Section 5.1

**INPUT BOXES**

Input boxes provide a simple way to gather input without placing a text box on a form.
Overview

- An input box provides a quick and simple way to ask the user to enter data
  
  - User types a value in the text box
  - OK button returns a string value containing user input
  - Cancel button returns an empty string
  - Should not be used as a primary method of input
  - Convenient tool for developing & testing applications
**InputBox(Prompt [,Title] [,Default] [,Xpos] [,Ypos])**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prompt</strong></td>
<td>String displayed in the input box, normally asks the user for a value</td>
</tr>
<tr>
<td><strong>Title</strong></td>
<td>String that appears in the title bar, contains project name by default</td>
</tr>
<tr>
<td><strong>Default</strong></td>
<td>String to be initially displayed in the text box, empty by default</td>
</tr>
<tr>
<td><strong>Xpos</strong></td>
<td>Integer that specifies the distance (in pixels) of the leftmost edge of the input box from the left edge of the screen, centered horizontally by default</td>
</tr>
<tr>
<td><strong>Ypos</strong></td>
<td>Integer that specifies the distance (in pixels) of the topmost edge of the input box from the top of the screen, placed near the top of the screen by default</td>
</tr>
</tbody>
</table>
Example Usage

• To retrieve the value returned by the **InputBox** function, use the assignment operator to assign it to a variable

• For example, the following statement assigns the string value returned by the **InputBox** function to the string variable **strUserInput**

```
Dim strUserInput As String = InputBox("Enter the distance.",
                                       "Provide a Value",
                                       "150")
```

• The string value that appears inside the text box will be stored in the **strUserInput** variable after the OK button is clicked and the input box closes
Section 5.2

LIST BOXES

List boxes display a list of items and allow the user to select an item from the list.
A **ListBox** control displays a list of items and also allows the user to select one or more items from the list
- Displays a scroll bar when all items cannot be shown

To create a ListBox control:
- Double-click the ListBox icon in the **Toolbox** window
- Position and resize the control as necessary

In **Design** mode, the list box appears as a rectangle
- The size of the rectangle determines the size of the list box

Use the **lst** prefix when naming a list box (**lstListBox**)

---

Copyright © 2011 Pearson Addison-Wesley
The **Items** Property

- The entries in a list box are stored in a property named **Items**
  - The **Items property** holds an entire list of values from which the user may choose
  - The list of values may be established at design time or runtime
  - Items are stored in a **Collection** called the **Items Collection**
Adding Items to the Items Collection

• To store values in the **Items** property at design time:
  – Select the **ListBox control** in the **Designer** window
  – In the **Properties** window, click the Items **(Collection)** ellipsis button (...)
  – Type each value on a separate line in the **String Collection Editor** dialog box
The **Items.Count** Property

- The **Items.Count** property returns the number of list box items or zero if the list is empty.
- For example, the **Items.Count** return value:
  - Can be used in an **If** statement:
    ```vbscript
    If lstEmployees.Items.Count = 0 Then
        MessageBox.Show("The list has no items!")
    End If
    ```
  - Or assigned to a variable
    ```vbscript
    IntNumEmployees = lstEmployees.Items.Count
    ```
• The **Items** property values can be accessed from your VB code
• Each item value is given a sequential index
  – The first item has an index of 0
  – The second item has an index of 1, etc.
• When assigning an item to a variable, you must explicitly convert the item to the same data type as the variable
  – Examples:

    ```vbnet
    strName  = lstCustomers.Items(2).ToString()
    intRoomNumber = CInt(lstRoomNumbers.Items(0))
    ```
Handling Exceptions Caused by Indexes

- An exception is thrown if an index is out of range
  - An exception handler can be used to trap indexing errors
    ```csharp
    Try
    strInput = lstMonths.Items(intIndex).ToString()
    Catch ex As Exception
        MessageBox.Show(ex.Message)
    End Try
    ```
  - Some programmers prefer to use an `If` statement to handle indexing errors
    ```csharp
    If intIndex >= 0 And intIndex < lstMonths.Items.Count Then
        strInput = lstMonths.Items(intIndex).ToString()
    Else
        MessageBox.Show("Index is out of range: " & intIndex)
    End If
    ```
The **SelectedIndex** property returns an integer with the index of the item selected by the user.

- If no item is selected, the value is set to -1 (an invalid index value).
- Can use **SelectedIndex** to determine if an item has been selected by comparing to -1.

**Example:**

```vbnet
If lstLocations.SelectedIndex <> -1 Then
    strLocation = lstLocations.Items(lstLocations.SelectedIndex).ToString()
End If
```
The **SelectedItem** Property

- The **SelectedItem** property contains the currently selected item from the list box
- For example:

```csharp
If lstItems.SelectedIndex <> -1
    strItemName = lstItems.SelectedItem.ToString()
End If
```
The Sorted Property

- **Sorted** is a Boolean property
- When set to **True**, values in the **Items** property are displayed in alphabetical order
- When set to **False**, values in the **Items** property are displayed in the order they were added
- Set to **False** by default
The `Items.Add` Method

- To store values in the Items property with code at runtime, use the `Items.Add` method
- Here is the general format:

  ```
  ListBox.Items.Add(Item)
  ```
- `ListBox` is the name of the `ListBox` control
- `Item` is the value to be added to the Items property
- Example:

  ```
  lstStudents.Items.Add("Sharon")
  ```
The **Items.Insert** Method

- To insert an item at a specific position, use the `Items.Insert` method.
- **General Format:**
  
  ```
  ListBox.Items.Insert(Index, Item)
  ```

- **ListBox** is the name of the `ListBox` control.
- **Index** is an integer value for the position where **Item** is to be placed in the **Items collection**.
- **Item** is the item you wish to insert.
- Items that follow are moved down.
- **For example:**
  
  ```
  lstStudents.Items.Insert(2, "Jean")
  ```
Methods to Remove Items

- `ListBox.Items.RemoveAt(Index)`
  - Removes item at the specified `Index`
- `ListBox.Items.Remove(Item)`
  - Removes item with value specified by `Item`
- `ListBox.Items.Clear()`
  - Removes all items in the Items property

Examples:

- `lstStudents.Items.RemoveAt(2)`  ' Remove 3rd item
- `lstStudents.Items.Remove("Jean")`  ' Remove item "Jean"
- `lstStudents.Items.Clear()`  ' Remove all items
Other List Box Methods

- **ListBox.Items.Contains(Item)**
  - Returns **True** if **Item** is found in the collection
- **ListBox.Items.IndexOf(Item)**
  - Returns an integer with the index position of the first occurrence of **Item** in the collection

Examples:

```vbnet
blnFound = lstMonths.Items.Contains("March")
intIndex = lstMonths.Items.IndexOf("March")
```

- Tutorial 5-1 provides more examples of list box controls, methods and properties
### Important Collection Methods and Properties

<table>
<thead>
<tr>
<th>Method or Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Add</strong>( <code>item As Object</code>)</td>
<td><strong>Method</strong>: adds <em>item to the collection</em>, returning its index position.</td>
</tr>
<tr>
<td><strong>Clear</strong>( )</td>
<td><strong>Method</strong>: removes all items in the collection. No return value.</td>
</tr>
<tr>
<td><strong>Contains</strong>( <code>value As Object</code>)</td>
<td><strong>Method</strong>: returns <em>True if value is found at least</em> once in the collection.</td>
</tr>
<tr>
<td><strong>Count</strong></td>
<td><strong>Property</strong>: returns the number of items in the collection. Read-only.</td>
</tr>
<tr>
<td><strong>IndexOf</strong>( <code>value As Object</code>)</td>
<td><strong>Method</strong>: returns the Integer index position of the first occurrence of <em>value in the collection</em>. If <em>value is not found</em>, the return value is –1.</td>
</tr>
<tr>
<td><strong>Insert</strong>( <code>index As Integer, item As Object</code>)</td>
<td><strong>Method</strong>: insert <em>item in the collection at position index</em>. No return value</td>
</tr>
<tr>
<td><strong>Item</strong>( <code>index As Integer</code>)</td>
<td><strong>Property</strong>: returns the object located at position <code>index</code>.</td>
</tr>
<tr>
<td><strong>Remove</strong>( <code>value As Object</code>)</td>
<td><strong>Method</strong>: removes <em>value from the collection</em>. No return value.</td>
</tr>
<tr>
<td><strong>RemoveAt</strong>( <code>index As Integer</code>)</td>
<td><strong>Method</strong>: removes the item at the specified <code>index</code>. No return value.</td>
</tr>
</tbody>
</table>
Section 5.3

INTRODUCTION TO LOOPS: THE DO WHILE LOOP

A loop is a repeating structure that contains a block of program statements.
Introduction

- A repetition structure, or loop causes one or more statements to repeat
- Each repetition of the loop is called an iteration

Visual Basic has three types of loops:
  - Do While
  - Do Until
  - For... Next

- The difference among them is how they control the repetition
The Do While Loop

- The **Do While** loop has two important parts:
  - a Boolean expression that is tested for a **True** or **False** value
  - a statement or group of statements that is repeated as long as the Boolean expression is true, called **Conditionally executed statements**

\[
\text{Do While BooleanExpression statement}
\]
\[
\text{(more statements may follow)}
\]
\[
\text{Loop}
\]

Expression true?  

Process

True

False
Example Do While Loop

- `intCount` initialized to 0
- Expression `intCount < 10` is tested
- If `True`, execute body:
  - "Hello" added to `lstOutput` Items Collection
  - `intCount` increases by 1
- Test expression again
  - Repeat until `intCount < 10` becomes `False`
Infinite Loops

- A loop must have some way to end itself
- Something within the body of the loop must eventually force the test expression to false
- In the previous example
  - The loop continues to repeat
  - `intCount` increases by one for each repetition
  - Finally `intCount` is not `< 10` and the loop ends
- If the test expression can never be false, the loop will continue to repeat forever
  - This is called an infinite loop
Counters

- A **counter** is a variable that is regularly incremented or decremented each time a loop iterates.

- **Increment** means to **add 1** to the counter’s value
  - \( \text{intX} = \text{intX} + 1 \)
  - \( \text{intX} += 1 \)

- **Decrement** means to **subtract 1** from the counter’s value
  - \( \text{intX} = \text{intX} - 1 \)
  - \( \text{intX} -= 1 \)

- Counters generally initialized before loop begins
  - ' **Start at zero**
    
    Dim intCount As Integer = 0

- Counter must be modified in body of loop
  - ' **Increment the counter variable**
    
    intCount += 1

- Loop ends when of value counter variable exceeds the range of the test expression
  - ' **False after ten iterations**
    
    intCount < 10
• Previous **Do While** loops are in **pretest** form
  – Expression is tested before the body of the loop is executed
  – The body may not be executed at all

• **Do While** loops also have a **posttest** form
  – The body of the loop is executed first
  – Then the expression is evaluated
  – Body repeats as long as expression is true
  – A posttest loop always executes the body of the loop at least once
The Posttest Do While Loop

• The **Do While** loop can also be written as a *posttest* loop:

  ```
  Do 
  Statement 
  (More statements may follow) 
  Loop While BooleanExpression
  ```

• **While BooleanExpression appears** after the **Loop** keyword
• Tests its Boolean expression after each loop iteration
• Will always perform at least one iteration, even if its Boolean expression is false to start with
Example Posttest Do While Loop

```
Dim intCount As Integer = 100
Do
    MessageBox.Show("Hello World!")
    intCount += 1
Loop While intCount < 10
```

- `intCount` is initialized to 100
- The statements in the body of the loop execute
- The expression `intCount < 10` is tested
- The expression is False
- The loop stops after the first iteration

- Tutorial 5-3 modifies Tutorial 5-2 to use a posttest **Do While** Loop
Keeping a Running Total

- Many programming tasks require you to calculate the total of a series of numbers
  - Sales Totals
  - Scores
- This calculation generally requires two elements:
  - A loop that reads each number in the series and accumulates the total, called a running total
  - A variable that accumulates the total, called an accumulator
Set accumulator to 0

Is there another number to read?

Read the next number

Add the number to the accumulator

Setting the accumulator variable to zero before entering the loop is a critical step

Yes (True)

No (False)
A Posttest Running Total Loop

- Tutorial 5-4 uses the code shown here in pretest form as part of a more complete example.
- Tutorial 5-5 demonstrates how to structure a loop such that the user can specify the iterations.

```vba
Const intNUM_DAYS = 5         ' Number days
Dim intCount As Integer = 1   ' Loop counter
Dim decSales As Decimal       ' Daily sales
Dim decTotal As Decimal = 0   ' Total sales
Dim strInput As String        ' Input string
' Get sales for each day.
Do
    ' Get daily sales amount from the user.
    strInput = InputBox("Enter the sales for day" & intCount.ToString())
    ' Convert user input string to a decimal.
    If Decimal.TryParse(strInput, decSales) Then
        decTotal += decSales ' Increment total
        intCount += 1        ' Input counter
    Else
        MessageBox.Show("Enter a number.")
    End If
Loop While intCount <= intNUM_DAYS
```
Section 5.4

THE DO UNTIL AND FOR...NEXT LOOPS

The **Do Until** loop iterates until its test expression is true. The **For...Next** loop uses a counter variable and iterates a specific number of times.
The Do Until Loop

• A **Do Until** loop iterates until an expression is true
  – Repeats as long as its test expression is **False**
  – Ends when its test expression becomes **True**
  – Can be written in either pretest or posttest form

  *Pretest General Format:*
  ```
  Do Until BooleanExpression
  Statement
  (More statements may follow)
  Loop
  ```

  *Posttest General Format:*
  ```
  Do
  Statement
  (More statements may follow)
  Loop Until BooleanExpression
  ```

• Tutorial 5-6 provides a hands-on example of a pretest **Do Until** loop
The **For...Next** Loop

- Ideal for loops that require a counter, pretest form only

  ```plaintext
  For CounterVariable = StartValue To EndValue [Step Increment]
  statement
  (more statements may follow)
  Next [CounterVariable]
  ```

- **For**, **To**, and **Next** are keywords
- **CounterVariable** tracks number of iterations
- **StartValue** is initial value of counter
- **EndValue** is counter number of final iteration
- Optional **Step Increment** allows the counter to increment at a value other than 1 at each iteration of the loop
Example of For...Next Loop

For intCount = 1 To 10
    MessageBox.Show("Hello")
Next

• Step 1: intCount is set to 1 (the start value)
• Step 2: intCount is compared to 10 (the end value)
  » If intCount is less than or equal to 10
    • Continue to Step 3
    • Otherwise the loop is exited
• Step 3: The MessageBox.Show("Hello") statement is executed
• Step 4: intCount is incremented by 1
• Step 5: Go back to Step 2 and repeat this sequence
Flowchart of For...Next Loop

- intCount <= 10?
  - Yes: Display "Hello"
  - No: Add 1 to intCount
- Set intCount to 1
Specifying a Step Value

• The **step** value is the value added to the counter variable at the end of each iteration
• Optional and if not specified, defaults to 1
• The following loop iterates 10 times with counter values 0, 10, 20, ..., 80, 90, 100

```vbnet
For intCount = 0 To 100 Step 10
    MessageBox.Show(intCount.ToString())
Next
```

• Step value may be negative, causing the loop to count downward

```vbnet
For intCount = 10 To 1 Step -1
    MessageBox.Show(intCount.ToString())
Next
```
• The **For...Next** loop can be used to calculate the sum of a series of numbers

```vba
Dim intCount As Integer ' Loop counter
Dim intTotal As Integer = 0 ' Accumulator
' Add the numbers 1 through 100.
For intCount = 1 To 100
    intTotal += intCount
Next
' Display the sum of the numbers.
MessageBox.Show("The sum of 1 through 100 is " & intTotal.ToString())
```
Breaking Out of a Loop

• In some cases it is convenient to end a loop before the test condition would end it

• The following statements accomplish this
  – **Exit Do** (used in **Do While** or **Do Until** loops)
  – **Exit For** (used in **For...Next** loops)

• Use this capability with caution
  – It bypasses normal loop termination
  – Makes code more difficult to debug
Deciding Which Loop to Use

- Each type of loop works best in different situations
  - The **Do While** loop
    - When you wish the loop to repeat as long as the test expression is true or at least once as a pretest loop
  - The **Do Until** loop
    - When you wish the loop to repeat as long as the test expression is false or at least once as a pretest loop
  - The **For...Next** loop
    - Primarily used when the number of required iterations is known
Section 5.5

NESTED LOOPS

A loop that is contained inside another loop is called a nested loop.
• A **nested loop** is a loop inside another loop
• The hands of a clock make a good example
  – The hour hand makes 1 revolution for every 60 revolutions of the minute hand
  – The minute hand makes 1 revolution for every 60 revolutions of the second hand
  – For every revolution of the hour hand the second hand makes 36,000 revolutions
Nested Loop Example

- The simulated clock example contains
  - An outer loop for the hours
  - A nested middle loop for the minutes
  - A nested inner loop for the seconds

For intHours = 0 To 23
  lblHours.Text = intHours.ToString()
For intMinutes = 0 To 59
  lblMinutes.Text = intMinutes.ToString()
    For intSeconds = 0 To 59
      lblSeconds.Text = intSeconds.ToString()
        Next
    Next
  Next
Nested Loop Example Analysis

• The innermost (seconds) loop will iterate 60 times for each iteration of the middle (minutes) loop
• The middle (minutes) loop will iterate 60 times for each iteration of the outermost (hours) loop
• 24 iterations of the outermost (hours) loop require:
  – 1,440 iterations of the middle (minutes) loop
  – 86,400 iterations of the innermost (seconds) loop

• An inner loop goes through all its iterations for each iteration of the outer loop
• Multiply iterations of all loops to get the total iterations of the innermost loop
A multicolumn list box displays items in columns with a horizontal scroll bar, if necessary. A checked list box displays a check box next to each item in the list. A combo box performs many of the same functions as a list box, and it can also let the user enter text.
Multicolumn List Boxes

- ListBox control has a **Multicolumn** property
  - Boolean property with default value of **False**
  - If set to **True**, entries can appear side by side
- Below, **ColumnWidth** is set to **30**
- Note the appearance of a horizontal scroll bar in this case
Checked List Boxes

• A form of ListBox with the list box properties and methods already discussed
• One item at a time may be selected but many items in a Checked List Box can be checked
• The CheckOnClick property determines how items may be checked
  – False - user clicks item once to select it, again to check it
  – True - user clicks item only once to both select it and check it
Finding the Status of Checked Items

- The **GetItemChecked** method determines if an item is checked by returning a Boolean value.

- **General Format:**
  
  $\text{CheckedListBox.GetItemChecked(Index)}$

  - Returns **True** if the item at **Index** has been checked.
  - Otherwise, returns **False**.
The following code counts the number of checked items:

```vbscript
Dim intIndex As Integer   ' List box index
Dim intCheckedCities As Integer = 0  ' To count the checked cities

' Step through the items in the list box, counting
' the number of checked items.
For intIndex = 0 To clbCities.Items.Count - 1
    If clbCities.GetItemChecked(intIndex) = True Then
        intCheckedCities += 1
    End If
Next

' Display the number of checked cities.
MessageBox.Show("You checked " & intCheckedCities.ToString() & " cities.")
```
Combo Boxes Similar to List Boxes

- Both display a list of items to the user
- Both have `Items`, `Items.Count`, `SelectedIndex`, `SelectedItem`, and `Sorted` properties
- Both have `Items.Add`, `Items.Clear`, `Items.Remove`, and `Items.RemoveAt` methods
- These properties and methods work the same with combo boxes and list boxes
Additional Combo Box Features

- A combo box also functions like a text box
- The combo box has a **Text** property
- The user may enter text into a combo box
- Or the user may select the text from a series of list box type choices
- In code, we use the *cbo* prefix when naming combo boxes
Combo Box Styles

• Simple Combo Box
  – List is always shown

• Drop-down Combo Box
  – List appears when user clicks down arrow
  – User can type text or select
• Drop-down List Combo Box
  – Behaves like a Drop-Down Combo Box, but the user may not enter text directly

• Tutorial 5-9 demonstrates the combo box
List Boxes versus Combo Boxes

• If restricting the user to select items listed
  – If empty space
    • Use list box
  – If limited space
    • Use drop-down list combo box

• If allowing user to select an item listed or enter an entirely new item
  – If empty space
    • Use simple combo box
  – If limited space
    • Use drop-down combo box
Section 5.7

RANDOM NUMBERS

Visual Basic provides tools to generate random numbers and initialize the sequence of random numbers with a random seed value.
The Random Object

- Random numbers are used in games and simulations to create random events
- Computers create **pseudo-random** numbers, which are not truly random
- To generate random numbers in Visual Basic, create a `Random` object reference variable
- For example:
  ```vba
  Dim rand As New Random
  ```
  - Creates a new `Random` object in memory called `rand`
  - The `rand` variable can be used to call the object’s methods for generating random numbers
The Next Method

• Once you have created a Random object, call its Next method to get a random integer number

\[
\text{intNum} = \text{rand.Next}()
\]

• Calling Next with no arguments
  – Generates an integer between 0 and 2,147,483,647

• Alternatively, you can specify an integer argument for the upper limit
  – The following Next method generates a number between 0 and 99

\[
\text{intNum} = \text{rand.Next}(100)
\]

• Numeric range does not have to begin at zero
  – Add or subtract to shift the numeric range upward or downward

\[
\text{intNum} = \text{rand.Next}(10) + 1
\]

\[
\text{intNum} = \text{rand.Next}(100) - 50
\]
The NextDouble Method

- Call a Random object’s NextDouble method to get a random floating-point number in the range of 0.0 up to (but not including) 1.0
  
  \[
  \text{dblNum} = \text{rand.NextDouble()}
  \]

- If you want the random number to fall within a larger range, multiply it by a scaling factor
  
  - The following statement generates a random number between 0.0 and 500.0
    
    \[
    \text{dblNum} = \text{rand.NextDouble()} \times 500.0
    \]
  
  - The following statement generates a random number between 100.0 and 600.0
    
    \[
    \text{dblNum} = (\text{rand.NextDouble()} \times 500.0) + 100.0
    \]

- Tutorial 5-10 uses random numbers to simulate a coin toss
• The **seed value** is used in the calculation that returns the next random number in the series
• Using the same seed value results in the same series of random numbers
• The system time, which changes every hundredth of a second, is the preferred seed value used by a **Random** object in most cases
• You can specify the seed value if you desire, when you create a **Random** object
• For example:

  ```vbnet
  Dim rand As New Random(1000)
  ```
  
  – **1000** as the seed value generates the same series of random numbers
  – Useful for specific tests and validations
  – Boring and repetitive for computer games or simulations
Section 5.8

SIMPLIFYING CODE WITH THE WITH...END WITH STATEMENT

The **With...End With** statement allows you to simplify a series of consecutive statements that perform operations using the same object.
• Multiple statements that use the same control or other object
  
  ```csharp
  txtName.Clear()
  txtName.ForeColor = Color.Blue
  txtName.BackColor = Color.Yellow
  txtName.BorderStyle = BorderStyle.Fixed3D
  ```

• Can be simplified using the `With...End With` statement
  
  ```csharp
  With txtName
    .Clear()
    .ForeColor = Color.Blue
    .BackColor = Color.Yellow
    .BorderStyle = BorderStyle.Fixed3D
  End With
  ```

•Eliminates the need to repeatedly type the control name
Section 5.9

TOOLTIPS

ToolTips are a standard and convenient way of providing help to the users of an application. The ToolTip control allows you to assign pop-up hints to the other controls on a form.
What is a Tool Tip?

- A **Tool Tip** is the short text message you see when holding the mouse over a control.
- These are easy to set up and use in Visual Basic forms.
- The **ToolTip control** allows you to create tool tips for other controls on a form.
Adding a **ToolTip** Control

- Display the form in *Design* view
- Double-click the **ToolTip** tool in the **Toolbox**
- The **ToolTip** control is invisible at runtime
  - It appears in the *component tray*, not the form
  - Component tray is a resizable region at the bottom of the *Design* window that hold invisible controls
- Form controls now have a **ToolTip property**
- This new property holds the text string that will be displayed for that control
ToolTip Properties

• Select the **ToolTip** control from the tray
• View **Properties** window to see the following
  – An **InitialDelay** property that regulates the delay before a tip appears
  – An **AutoPopDelay** property that determines how long a tip is displayed
  – The **ReshowDelay** property determines the time between the display of different tips as the user moves the mouse from control to control
• Tutorial 5-11 demonstrates adding tool tips to a form
In this section, you build the *Vehicle Loan Calculator* application. The application uses a loop, input validation, and ToolTips. This section also covers some of the Visual Basic intrinsic financial functions.
Introduction

- Visual Basic has several built-in functions for performing financial calculations
- You will build a program named *Vehicle Loan Calculator*
- It uses the following functions:
  - `Pmt`
  - `Ipmt`
  - `PPmt`
The `Pmt` Function

- The `Pmt` function returns the periodic payment amount for a loan with a fixed interest rate.

\[
Pmt(\text{PeriodicInterestRate}, \text{NumberOfPeriods}, -\text{LoanAmount})
\]
- `PeriodicInterestRate` is the rate of interest per period.
- `NumberOfPeriods` is the total number of months.
- `LoanAmount` is the amount being borrowed, must be negative.

- For example:

\[
dblPayment = Pmt(dblAnnInt / 12, 24, -5000)
\]
- `dblAnnInt` holds the annual interest rate.
- `24` is the number of months of the loan.
- The amount of the loan is $5000.
- `dblPayment` holds the fixed monthly payment amount.
The IPmt Function

- The IPmt function returns the interest payment for a specific period of a loan with a fixed interest rate and fixed monthly payments

$$\text{IPmt} (\text{PeriodicInterestRate}, \text{Period}, \text{NumberOfPeriods}, -\text{LoanAmount})$$

- PeriodicInterestRate is the rate of interest per period
- Period is the period for which you would like the payment
- NumberOfPeriods is the total number of months
- LoanAmount is the amount being borrowed, must be negative

- For example:

  $$\text{dblInterest} = \text{IPmt} (\text{dblAnnInt} / 12, 6, 24, -5000)$$

  - dblAnnInt holds the annual interest rate
  - 6 is the number of the month for which to calculate the payment
  - 24 is the number of months of the loan
  - The amount of the loan is $5000
  - dblInterest holds the amount of interest paid in month 6 of the loan
• The **PPmt function** returns the principal payment for a specific period on a loan with a fixed interest rate and fixed monthly payments

\[
\text{PPmt} (\text{PeriodicInterestRate}, \text{Period}, \text{NumberOfPeriods}, -\text{LoanAmount})
\]

• *PeriodicInterestRate* is the rate of interest per period
• *Period* is the period for which you would like the payment
• *NumberOfPeriods* is the total number of months
• *LoanAmount* is the amount being borrowed, must be negative

• For example:

```
dblPrincipal = PPmt(dblAnnInt / 12, 6, 24, -5000)
```

• *dblAnnInt* holds the annual interest rate
• 6 is the number of the month for which to calculate the payment
• 24 is the number of months of the loan
• The amount of the loan is $5000
• *dblPrincipal* holds the amount of principal paid in month 6 of the loan
A credit union branch manager asks you to write an application named *Vehicle Loan Calculator* that displays the following information for a loan:

- The monthly payment amount
- The amount of the monthly payment applied toward interest
- The amount of the monthly payment applied toward principal

The credit union currently charges

- 8.9% annual interest for new vehicle loans
- 9.5% annual interest on used vehicle loans
Sketch of the Vehicle Loan Calculator Form
## Event Handlers

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>btnCalculate_Click</strong></td>
<td>Calculates and displays a table in the list box showing interest and principal payments for the loan</td>
</tr>
<tr>
<td><strong>btnClear_Click</strong></td>
<td>Resets the interest rate, clears the text boxes, and clears the list box</td>
</tr>
<tr>
<td><strong>btnExit_Click</strong></td>
<td>Ends the application</td>
</tr>
<tr>
<td><strong>radNew_CheckedChanged</strong></td>
<td>Updates the annual interest rate if the user selects a new vehicle loan</td>
</tr>
<tr>
<td><strong>radUsed_CheckedChanged</strong></td>
<td>Updates the annual interest rate if the user selects a used vehicle loan</td>
</tr>
</tbody>
</table>
The pseudocode does not indicate input validation, and the actual arguments that need to be passed to the `Pmt`, `IPmt`, and `PPmt` functions are not shown.

Get VehicleCost from the form
Get DownPayment from the form
Get Months from the form
Loan = VehicleCost – DownPayment
MonthlyPayment = Pmt()
For Count = 0 To Months
    Interest = IPmt()
    Principal = PPmt()
    Display Month, Payment, Interest, and Principal in list box
Next
**btnCalculate_Click Event Handler Flowchart**

1. **Start**
   - Get the number of months
   - Loan = vehicle cost – down payment
   - Calculate the monthly payment
   - count = 1
   - A

2. **count > number of months?**
   - If Yes, go to End
   - If No, go to Calculate the interest for the period

3. **Calculate the interest for the period**

4. **Calculate the principal for the period**

5. **Display the month, payment, interest, and principal in the list box**

6. **Add 1 to count**

---

Copyright © 2011 Pearson Addison-Wesley  
Chapter 5 – Slide 78
radNew_CheckedChanged & radUsed_CheckedChanged

Event Handler Pseudocode

If radNew is selected Then
    Annual Interest Rate = 0.089
    Display Annual Interest Rate in lblAnnInt
End If

If radUsed is selected Then
    Annual Interest Rate = 0.095
    Display Annual Interest Rate in lblAnnInt
End If
Event Handler Pseudocode

**radNew_CheckedChanged & radUsed_CheckedChanged**

### radNew_CheckedChanged
- **Start**
- **radNew radio button selected?**
  - **Yes**
    - annual interest rate = 0.089
  - **No**
    - Copy "8.9%" to lblAnnInt
- **End**

### radUsed_CheckedChanged
- **Start**
- **radUsed radio button selected?**
  - **Yes**
    - annual interest rate = 0.095
  - **No**
    - Copy "9.5%" to lblAnnInt
- **End**