Additional Controls, Scope, Random Numbers, and Making Decisions
CS109

In this lecture we will briefly examine a few new controls and how to make if-then-else statements. The textbook covers most of these in chapter 4.

Combo Box

The Combo Box control is like a textbox with a pull-down menu of choices. We can access the user's selection with:

```
comboBox.Text
```

We can add to the items just like with a listbox:

```
comboBox.Items.Add(newItem)
```

(Short demo in class)

Group Box Control

The group box is used to group related sets of controls for visual effect. To use it, drag the group box onto the form. Then drag any new controls into the group box. The new controls will now be "part" of the group box.

The group box can be used to create different sets of radio buttons (upcoming).

(Short demo in class)

Check Box Control

The checkbox is a small box that can be checked or unchecked by the user. To see if something is checked or not you can inspect the "Checked" Property:

```
checkbox.Checked            - True if checked, False if not
```

(Short demo in class)

Radio Button Control

The radio button operates like an old car radio. When one button is pushed, any other buttons "pop out". For all radio buttons that are on a form, only one can be active at a time. If you would like to have multiple subgroups of radio buttons then they should be added to a GroupBox.

To see the value of a radio button, you can inspect the .Checked property just as with a checkbox.

(Short demo in class)

Main Menu Control

This control allows you to add a menu to the application. To use it, drag a Main Menu control to your form. Then double-click it in the form area. A menu designer will appear at the top of your form saying "Type Here".

You can now type the name of the top-level of your menu. Click and type to fill in sub-areas. To attach code to the sub-areas, double-click on the menu item. The VB Code window will appear with an event for you to fill in code.

Try adding a menu for F)ile, O)pen, and C)lose.

Variable Scope

Variable scope was described in the previous chapter, but we'll give a brief review of this concept again here because it is very important to understand.

Scope refers to the section of code where a variable is "alive". There are two categories of scope for a variable in VB.NET that are generally used: *class* or *module* or *global* scope, and *local* scope. Both adhere to the same basic rule: **a variable is accessible everywhere within the form or subroutine where it is declared, including code within nested subroutines**.

Local Variables

A local variable only has scope within the subroutine it is created. When a variable is declared within a subroutine procedure with a Dim statement, space reserved in memory for that variable exists until the End Sub. After the subroutine exits, the variable ceases to exist

```
Private Sub Button1_Click(...) Handles Button1.Click
   Dim num As Integer = 3       // Local Variable, reset to 3 when run
   num = num + 5
   MessageBox.Show(num)         // Shows 8 every time
End Sub
```

Your program is free to have other subroutines that use the variable "num" and each will refer to a different number.

Class or Module or Global Variables

Class or module variables are visible to every procedure in a form's code. Dim statements for Class-Level variables are placed outside all procedures at the top of the program region. This is useful for variables you would like to use within many procedures on the form.

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    Dim strName As String       ' Class or Module Level Variables

    Private Sub Button1_Click(...) Handles Button1.Click
            strName = InputBox("Enter your name")
        End Sub

    Private Sub Button2_Click(...) Handles Button2.Click
            Console.WriteLine("Your name is " & strName)
        End Sub
    End Class

End Class
```

This technique is also a way in which subroutines can send data to each other – one subroutine can set the class level variable while the other reads it.

In normal usage, variable scoping is as simple as defined above. However, things get trickier when variables have the same name. For example, consider the following scenario:

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    ' Class-level variable set to "Hello"
    Dim strName As String = "Hello"

    Private Sub Button1_Click(...) Handles Button1.Click
        ' Local variable set to "There"
        Dim strName As String = "There"
        Console.WriteLine(strName)      ' Outputs "There"
        ' Local variable takes precedence over class-level variable
    End Sub
End Class
```

In the example above we have two variables named strName. One has class scope, the other has local scope. Which variable is referenced when there is this ambiguity?

The rule used in VB.NET is that the scope of a variable begins with its most recent declaration. This means that local variables take precedence over class variables. In the example above, the local variable is output. Similarly, if we changed the variable strName inside the Button1_Click event, we would change the local variable while the class variable remains unchanged.

If we wanted to access the class or module level variable, use the keyword “Me” in front of the variable name:

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    ' Class-level variable set to "Hello"
    Dim strName As String = "Hello"

    Private Sub Button1_Click(...) Handles Button1.Click
        ' Local variable set to "There"
        Dim strName As String = "There"
        Console.WriteLine(Me.strName)       ' Outputs "Hello"
    End Sub
End Class
```

It is common convention to always use the Me prefix for class level variables. Once again, this example only demonstrates output of a variable, but we could also assign the class or local variable to a different value.

VB.NET does not allow us to have multiple local variables with the same name within the same scope. The following is illegal;

```
    Private Sub Button1_Click(...) Handles Button1.Click
        Dim strName As String = "There"

        Console.WriteLine(strName)

        Dim strName As String = "New String"     ‘ ILLEGAL REDEFINITION

        Console.WriteLine(strName)
    End Sub
```

Random Numbers

Random number generation is described on page 529 of the book. It is often useful to generate random numbers to produce simulations or games (or homework problems :) One way to generate these numbers in VB.NET is to use the method **Random** object.

The random object generates pseudo-random numbers. What is a pseudo-random number? It is a number that is not truly random, but appears random. That is, every number between 0 and 1 has an equal chance (or probability) of being chosen each time random() is called. (In reality, this is not the case, but it is close).

Here is a very simple pseudorandom number generator to compute the ith random #:

$R_i = (R_{i-1} * 7) \bmod 11$

Initially, we set the “seed”, $R_0 = 1$. Then our first “random” number is 7 mod 11 or 7. Our second “random” number is then (7*7) mod 11 or 5.

Our third "random" number is then (5*7) mod 11 or 3.
Our fourth "random" number is then (3*7) mod 11 or 10.
..etc.

As you can see, the values we get seem random, but are really not. This is why they are called pseudorandom. We can get slightly more random results by making the initial seed some variable number, for example, derived from the time of day. The particular function shown above would not be a very good pseudorandom number generator because it would repeat numbers rather quickly.

Here is an example of using VB.NET's random number generator.

1. At the class level, create a variable of type Random:

   ```
   Dim rnd As New Random
   ```

   This creates a new Random object. We'll talk more about objects later when we get to object-oriented programming. It is important to define this as a class level variable, or you won't get a good pseudorandom number sequence.

2. To generate a random integer *x*, where $min \leq x < max$, use:

   ```
   x = rnd.Next(min, max)
   ```

3. To generate a random double *d*, where $0 \leq d < 1$, use:

   ```
   d = rnd.NextDouble()
   ```

Here is a short demonstration program:

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    Dim rnd As New Random                  ' Create rnd object at class level

    Private Sub Button1_Click(...) Handles Button1.Click
        Dim intNum As Integer
        Dim dbl As Double

        intNum = rnd.Next(0, 4)
        Console.WriteLine(intNum)

        dbl = rnd.NextDouble()
        Console.WriteLine(dbl)
    End Sub
End Class
```

The program above might print out:

```
1
0.969432235215526
```

The second time the button is clicked it might print out:

```
3
0.032952289578017
```

Both the integer and double are randomly generated. While the method call allows us to specify the range for integers, what if instead we wanted a random double between 5 and 15? We can just invoke rnd with:

```
	intNum = rnd.Next(5, 16)
```

This generates a number that is ≥ 5 and < 16 (i.e. 5-15 inclusive).

Boolean Expressions and Conditions

The physical order of a program is the order in which the statements are *listed.* The logical order of a program is the order in which the statements are *executed.* With conditional structures and control structures that we will examine soon, it is possible to change the order in which statements are executed.

Boolean Data Type

To ask a question in a program, you make a statement. If your statement is true, the answer to the question is yes. If your statement is not true, the answer to the question is no. You make these statements in the form of *Boolean expressions*. If you recall from the previous lecture, a Boolean expression asserts (states) that something is true. The assertion is evaluated and if it is true, the Boolean expression is true. If the assertion is not true, the Boolean expression is false.

In VB.NET, the data type **Boolean** is used to represent Boolean data. Each **boolean** constant or variable can contain one of two values: **True** or **False**.

Relational Operators

A Boolean expression can be a simple Boolean variable or constant or a more complex expression involving one or more of the relational operators. Relational operators take two operands and test for a relationship between them. The following table shows the relational operators and the VB.NET symbols that stand for them.

VB.NET Symbol	*Relationship*
=	Equal to
<>	Not equal to (since there is no ≠ symbol on the keyboard)
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

For example, the Boolean expression

```
number1 < number2
```

is evaluated to **True** if the value stored in **number1** is less than the value stored in **number2**, and evaluated to **False** otherwise.

Examples:

```
Dim b as Boolean
b = 3 < 1
Console.WriteLine(b)        ‘ Outputs “false”
b = 3 > 1
Console.WriteLine(b)        ‘ Outputs “true”
```

When a relational operator is applied between variables of type **String**, the assertion is in terms of where the two operands fall in the collating sequence of a particular character set. For example,

```
character1 < character2
```

is evaluated to **true** if the character stored in **character1** comes before the character stored in **character2** in the collating sequence of the machine on which the expression is being evaluated. Although the collating sequence varies among machines, you can think of it as being in alphabetic order. That is, *A* always comes before *B* and *a* always before *b*, but the relationship of *A* to *a* may vary. This is an artifact of the way the alphabet was defined in the ASCII code. For ASCII, it turns out the A < a.

```
Dim b as Boolean
b = “a” < “b”
Console.WriteLine(b)        ‘ Outputs “true”, ASCII a = 96, b = 97
b = “abc” < “BAD”
Console.WriteLine(b)        ‘ Outputs “false”, ASCII a = 96, B=66
```

We must be careful when applying the relational operators to floating point operands, such as doubles, particularly equal (=) and not equal (<>). Integer values can be represented exactly; floating point values with fractional parts often are not exact in the low-order decimal places. Therefore, you should compare floating point values for near equality. For now, *do not compare floating point numbers for equality*. Instead compare to a data range of interest.

Note that the relational operators are either binary; they take only two values. The following is not a valid way to see if n is between 2 and 5:

INVALID: 2 < n < 5 The accepted way is to use Boolean operators

Boolean Operators

A simple Boolean expression is either a Boolean variable or constant or an expression involving the relational operators that evaluates to either true or false. These simple Boolean expressions can be combined using the logical operations defined on Boolean values. There are three Boolean operators: AND, OR, and NOT. Here is a table showing the meaning of these operators and the symbols that are used to represent them in VB.NET.

VB.NET Keyword	*Meaning*
AND	AND is a binary Boolean operator. If both operands are true, the result is true. Otherwise, the result is false. **True** **False** **True** True False **False** False False
OR	OR is a binary Boolean operator. If at least one of the operands is true, the result is true. Otherwise, the result is false. **True** **False** **True** True True **False** True False

NOT	NOT is a unary Boolean operator. NOT changes the value of its operand: If the operand is true, the result is false; if the operand is false, the result is true. **Not_Value** **True** False **False** True

If relational operators and Boolean operators are combined in the same expression the Boolean operator NOT has the highest precedence, the relational operators have next higher precedence, and the Boolean operators AND and OR come last (in that order). Expressions in parentheses are always evaluated first.

For example, given the following expression (**stop** is a **boolean** variable)

```
count <= 10 and sum >= limit or not stop
```

not stop is evaluated first, the expressions involving the relational operators are evaluated next, the AND is applied, and finally the OR is applied.

It is a good idea to use parenthesis to make your expressions more readable, e.g:

```
(((count <=10) AND (sum>=limit))  OR  (NOT (stop)))
```

This also helps avoid difficult-to-find errors if the programmer forgets the precedence rules.

A common error is to replace the condition

```
Not ( 2 < 3 )
```

by the condition

```
( 2 > 3 )
```

The correct replacement is (2 >= 3) because >= is the opposite of <, just as <= is the opposite of >

Exercises: Are the following statements true or false?

```
Dim a as Integer = 2
Dim b as Integer = 3

3*a = 2*b

(a<b) Or (b<a)

(2<a) And (a<5)

Not ((a<b) And (a<(b+a)))

((a=b) And (a*a<b*b)) Or ((b<a) And (2*a<b))

“Car”<”Train”

“99”>”ninety-nine”

“9W” > “9a”

((“Ant” < “hill”) And (“mole” > “hill”)) Or Not (Not (“Ant” < “hill”) Or
        Not (“Mole” > “hill”))
```

If-Then and If-Then-Else Statements

The If statement allows the programmer to change the logical order of a program; that is, make the order in which the statements are executed differ from the order in which they are listed in the program. The If-Then statement uses a Boolean expression to determine whether to execute a statement or to skip it. The format is as follows:

```
If (boolean_expression1)
        statement1                  ‘ Expr1 true
ElseIf (boolean_expression2)
        statement2                  ‘ Expr1 false, Expr2 true
ElseIf (boolean_expression3)
        statement3                  ‘ Expr1, Expr2 false, Expr3 true
…
Else
        statement_all_above_false   ‘ Expr1, Expr2, Expr3 false
End If
```

The Else and ElseIf portions are optional. If you like you can leave them off. You can also insert multiple statements into each section if you have more than one line of code you would like to execute for each block.

Here are some examples of if statements.

To find the larger of two numbers:

```
Dim dblNum1, dblNum2, dblLargerNum As Double
dblNum1 = CDbl(txtFirstNum.Text)
dblNum2 = CDbl(txtSecondNum.Text)
If dblNum1 > dblNum2 Then
  dblLargerNum = dblNum1
Else
  dblLargerNum = dblNum2
End If
txtResult.Text = "The larger number is " & dblLargerNum
```

Setting an appropriate message for profit/loss:

```
If costs = revenue Then
  txtResult.Text = "Break even"
Else
  If costs < revenue Then
    profit = revenue - costs
    txtResult.Text = "Profit is " & FormatCurrency(profit)
  Else
    loss = costs - revenue
    txtResult.Text = "Loss is " & FormatCurrency(loss)
  End If
End If
```

Checking an answer for how much a ten gallon hat holds:

```
Dim dblAnswer As Double
dblAnswer = CDbl(txtAnswer.Text)
If (dblAnswer >= 0.5) And (dblAnswer <= 1) Then
  txtSolution.Text = "Good, "
Else
  txtSolution.Text = "No, "
End If
txtSolution.Text &= "it holds about 3/4 of" _
                  & " a gallon."
```

Code that takes as input a number between 0-100 and outputs a letter grade, where 90-100 is A, 80-90 is B, 70-80 is C, 60-70 is D, and anything below 60 is an F.

```
numGrade = CDbl(textBoxGrade.Text)
If (numGrade >= 90) Then
    Console.WriteLine("A")
ElseIf (numGrade >= 80) Then
    Console.WriteLine("B")
ElseIf (numGrade >= 70) Then
   Console.WriteLine("C")
ElseIf (numGrade >= 60) Then
   Console.WriteLine("D")
Else
    Console.WriteLine("F")
End If
```

Note that anything can go inside the body of the If statement – including other If statements! When we do this, it is called nested If statements. For example:

```
If (numGrade >= 90) Then
    If (numGrade > 96) Then
        Return "A+"
    ElseIf (numGrade > 93) Then
        Return "A"
    Else
        Return "A-"
    End If
ElseIf (numGrade >= 80) Then
    Return "B"
End If
```

In general, any Nested If statement can be turned into a single If statement using AND's as follows:

```
If cond1 Then
  If cond2 Then
    action
  End If
End If
```

```
If cond1 And cond2 Then
  action
End If
```

Nested If

Less Confusing

The format on the right is generally less confusing, although there are exceptions.

In-Class Exercise: Write a program that gives a short quiz about UAA:

1. Who is the current Dean of the College of Arts & Sciences?
    a. Kerry Feldman
    b. Jim Liszka
    c. Theodore Kassier
2. This is the most popular major at UAA, in terms of number of students enrolled in the program, in 2005-2006?
    a. Biological Sciences[1]
    b. Accounting
    c. Nursing
3. Total value of grants and contracts awarded to UAA (2002-03)?
    a. 4.9 million
    b. 20.1 million
    c. 30.5 million

At the end of the quiz, display the score of the test-taker, where 1 point is awarded for each correct question.

In-Class Exercise: Write a program that takes a year and determines if it is a leap year. Every year divisible by four is a leap year, with the **exception** of years divisible by 100 and not divisible by 400. For example:

1600 is a leap year: Divisible by 4, Divisible by 100, and Divisible by 400
2000 is a leap year: Divisible by 4, Divisible by 100, and Divisible by 400
1984 is a leap year: Divisible by 4, Not divisible by 100, Not divisible by 400
1700 is not a leap year: Divisible by 4, Divisible by 100, but not divisible by 400

Participatory In-Class Exercise:

Gambler's Dilemma: We will run this activity in class. Each student will be given a real dollar bill and a blank piece of paper.

The rules of this game are as follows. If all students "cooperate" then everyone keeps their dollar and goes home a little bit richer. If one student "defects" then that one student gets ALL the money. However, if more than one student "defects" then the instructor gets all the money back.

Each student will secretly write "cooperate" or "defect" on the note along with their name (in case of a single defector). After everyone has written their answer we will collect the notes and determine the outcome.

[1] Biological Sciences: 321, Accounting: 410, Nursing: 260

This activity is a multi-participant version of the Prisoner's Dilemma. The dilemma is the conflict between individual and group rationality. A group whose members pursue rational self-interest may all end up worse off than a group whose members act contrary to rational self-interest.

In the standard Prisoner's Dilemma, there are only two participants. From wikipedia:

> Two suspects, A and B, are arrested by the police. The police have insufficient evidence for a conviction, and, having separated both prisoners, visit each of them to offer the same deal: if one testifies for the prosecution against the other and the other remains silent, the betrayer goes free and the silent accomplice receives the full 10-year sentence. If both stay silent, the police can sentence both prisoners to only six months in jail for a minor charge. If each betrays the other, each will receive a two-year sentence. Each prisoner must make the choice of whether to betray the other or to remain silent. However, neither prisoner knows for sure what choice the other prisoner will make. So the question this dilemma poses is: What will happen? How will the prisoners act?

The dilemma can be summarised thus:

	Prisoner B Stays Silent	**Prisoner B Betrays**
Prisoner A Stays Silent	Both serve six months	Prisoner A serves ten years Prisoner B goes free
Prisoner A Betrays	Prisoner A goes free Prisoner B serves ten years	Both serve two years

This scenario appears in many natural settings, including economics, sociology, and biology. See http://www.brembs.net/ipd/ipd.html for some examples.

Write a program that allows a human to play against a computer in the Prisoner's Dilemma game. The computer should use the "tit for tat" strategy – cooperate (i.e. stay silent) the first time, and thereafter do the same thing that the player did last time. The program should count up the number of years served in prison by the human and the computer over repeated iterations of the game.

Select Case Blocks

A Select Case block a more compact way to construct what is equivalent to an if-then-elseif statement. Select statements use the value of a single expression called the **selector**. Possible actions are executed depending on the value of the selector.

The general format of a select block is:

```
Select Case selector
  Case valueList1
    action1
  Case valueList2
    action2
  Case Else
    action of last resort
End Select
```

The Case Else is optional.

Here is an example of using a select statement to take the finishing position of a horse and indicate if the outcome is Win, Place, or Show:

```
Dim position As Integer
position = CInt(txtPosition.Text)
Select Case position             ' position is the selector
  Case 1
    txtOutcome.Text = "Win"
  Case 2
    txtOutcome.Text = "Place"
  Case 3
    txtOutcome.Text = "Show"
  Case 4, 5
    txtOutcome.Text = "You almost placed in the money."
  Case Else
    txtOutcome.Text = "Out of the money."
End Select
```

Note that we can insert lists of values. For example, 4 and 5 are separated by a comma. If the user entered either 4 or 5 then txtOutcome.Text would be set to "You almost placed in the money."

We can specify data ranges and also use relational operators as show below:

```
Dim position As Integer
position = CInt(txtPosition.Text)
Select Case position
  Case 1 To 3
    txtOutcome.Text = "In the money."
  Case Is >= 4
    txtOutcome.Text = "Not in the money."
End Select
```

In-Class Exercise:

Write a program that inputs a number between 0 and 99 and outputs the value in English, e.g. “ninety nine” for 99, “zero” for 0, etc. Don’t use 100 different WriteLine statements!

Larger In-Class Exercise: The Monty Hall problem

You are a contestant on a game show and have won a shot at the grand prize. Before you are three doors. Behind one door is a new Mustang convertible and $1,000,000 in cash. Behind the other two doors are the booby prizes of macaroni & cheese plus a bottle of dishwasher detergent. The location of the prizes is randomly selected. You want the car and the cash. The game show host asks you to select a door, and you randomly pick one. However, before revealing the contents behind your door, the game show host reveals one of the other doors that contains the booby prize. At this point, the game show host asks if you would like to stick with your original choice or switch your choice to the remaining door. What choice should you make to optimize your chances of winning the grand prize, or does it matter?

Write a computer program to simulate one run of the Monty Hall problem. Make three buttons to represent the three doors. Pick a random number from 1-3 to represent the door that holds the grand prize. Let the user choose one of the doors. The program should then display on the button one of the doors that has the booby prize. Let the user click a button again, and this time display what is behind the door the user selected. Run the program several times – is it better to switch, or does it matter?

Pseudocode:

```
1. Set class level variable, FirstPick, to true. If true, this means the player is picking
   a door for the first time. If false, this means the player is picking the door after
   one door has been revealed so we should display what prize the player gets.
2. Create class level variable to generate random numbers
3. Create class level variable to store the door with the prize
4. In the form-load event (executed when the program first runs) set the prize to a
   random number from 1-3
5. Make three buttons on the form
6. In the button click event for button 1 (door 1)
   a. If FirstPick=True then
      i. See if door 2 is a booby prize. If so, display it. Otherwise see if
         door 3 is a booby prize. If so, display it.
      ii. Set FirstPick to False
   b. Else
      i. If the variable with the prize is 1, then display “You win the car”
         else display “You win detergent”
      ii. Reset FirstPick to True, reset button text, and pick a new door for
          the prize so we can play again
```