Chapter 4 General Procedures

- 4.1 Sub Procedures, Part I
- 4.2 Sub Procedures, Part II
- 4.3 Function Procedures
- 4.4 Modular Design

4.1 Sub Procedures, Part I

- Sub Procedures
- Calling Other Sub Procedures

Procedures

- So far, most of the code has been inside a single method for an event
  - Fine for small programs, but inconvenient for large ones
  - Much better to divide program into manageable pieces
- Benefits of modularization
  - Avoids repeat code (reuse a function many times in one program)
  - Promotes software reuse (reuse a function in another program)
  - Promotes good design practices (Specify function interfaces)
  - Promotes debugging (can test an individual module to make sure it works properly)

Devices for modularity

- VB.NET has two devices for breaking problems into smaller pieces:
  - Sub procedures
    - Short for ‘Subroutine’ or ‘Subprogram’
    - Executes a block of statements, returns no value
  - Function procedures
    - Identical to a sub, except returns a value

Sub Procedures

- Performs one or more related tasks
- General syntax, code goes inside the class for the form:

Sub *ProcedureName()*
statements
End Sub

Calling a Sub procedure

- The statement that invokes a Sub procedure is also referred to as a call statement
- A call statement looks like this:

ProcedureName()

You've already been using this for pre-defined functions, like Trim()

Naming Sub procedures

- The rules for naming Sub procedures are the same as the rules for naming variables.

Example

```
lstResult.Items.Clear()
ExplainPurpose()
lstResult.Items.Add("")

Sub ExplainPurpose()
    lstResult.Items.Add("This program displays a sentence")
    lstResult.Items.Add("identifying two numbers and their sum.")
End Sub
```

Code Re-Use

- If in another place in the code you wanted to explain the purpose, you can just invoke the subroutine:

```
Sub OtherCode(…)
        ExplainPurpose()
        ‘ Presumably other code here
End Sub
```

- Avoids duplicate the same code in many places
- If you ever want to change the code, only one place needs to be changed

Passing

- You can send items to a Sub procedure

```
Sum(2, 3)
```

```
Sub Sum(num1 As Double, num2 As Double)
    Console.WriteLine(num1+num2)
End Sub
```

- In the Sum Sub procedure, 2 will be stored in num1 and 3 will be stored in num2 and the sum will be output to the console

Passing

- We can pass variables too:

```
x = 2
y = 3
Sum(x,y)          ‘ Same as Sum(2, 3)
```

- The variables are evaluated prior to calling the subroutine, and their values are accessible via the corresponding variable names in the sub

Population Density Sub

- Subroutine to calculate population density:

```
Sub CalculateDensity(ByVal state As String, _
                     ByVal pop As Double, _
                     ByVal area As Double)
   Dim rawDensity, density As Double
   rawDensity = pop / area
   density = Math.Round(rawDensity, 1)     ' Round to 1 decimal place
   Console.Write("The density of " & state & " is " & density)
   Console.WriteLine(" people per square mile.")
End Sub
```

Parameters and Arguments

```
CalculateDensity("Alaska", 627000, 591000)
```

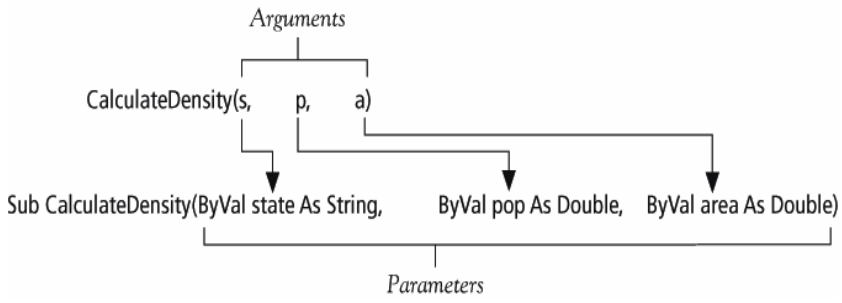
Arguments – what you send to a Sub procedure

```
Sub CalculateDensity(ByVal state As String, _
                     ByVal pop As Double, _
                     ByVal area As Double)
```

Parameters – place holders for what the sub procedure receives

If ByVal left off, VB.NET will add it

Figure 4.2



Code Reuse

- By making CalculateDensity a procedure subroutine, we can reuse it, e.g.:

  CalculateDensity(“Hawaii”, 1212000, 6471)

Sub Procedures Calling Other Sub Procedures

```
Private Sub btnDisplay_Click(...)
    Handles btnDisplay.Click
  FirstPart()
  Console.WriteLine(“a”)
End Sub

Sub FirstPart()
  SecondPart()
  Console.WriteLine(“b”)
End Sub

Sub SecondPart()
  Console.WriteLine(“c”)
End Sub
```

Output:

c
b
a

In Class Exercise

- Write a Sub procedure that takes as arguments an animal and sound for the “Old McDonald Had A Farm” song and outputs the verse, e.g.:
  - Old McDonald had a farm, E-I-E-I-O.
  - And on his farm he had a cow, E-I-E-I-O.
  - With a moo moo here, and a moo moo there,
  - Here a moo, there a moo, everywhere a moo moo.
  - Old McDonald had a farm, E-I-E-I-O
- Complete the program in the Form Load event to output the verses for a cow, chicken, and lamb.

4.2 Sub Procedures, Part II

- Passing by Value
- Passing by Reference
- Local Variables
- Class-Level Variables
- Debugging

Passing by Value

- ByVal stands for “By Value”
- ByVal parameters retain their original value after Sub procedure terminates

ByVal Example

```
Sub CallingSub()
    Dim y As Integer
    y = 5
    Console.WriteLine("y is " & y)
    ValSub(y)
    Console.WriteLine("y is " & y)
End Sub

Sub ValSub(ByVal x As Integer)
    x = 10
    Console.WriteLine(" x is " & x)
End Sub
```

Output?

ByVal Example – Y to X

```
Sub CallingSub()
    Dim x As Integer
    x = 5
    Console.WriteLine(“x is " & x)
    ValSub(x)
    Console.WriteLine(“x is " & x)
End Sub

Sub ValSub(ByVal x As Integer)
    x = 10
    Console.WriteLine("x is " & x)
End Sub
```

Output?

Passing by Reference

- ByRef stands for "By Reference"
- ByRef parameters can be changed by the Sub procedure and retain the new value after the Sub procedure terminates

ByRef Example

```
Sub CallingSub()
    Dim y As Integer
    y = 5
    Console.WriteLine("y is " & y)
    RefSub(y)
    Console.WriteLine("y is " & y)
End Sub

Sub RefSub(ByRef x As Integer)
    x = 10
    Console.WriteLine(" x is " & x)
End Sub
```

Output?

ByVal Example – Y to X

```
Sub CallingSub()
    Dim x As Integer
    x = 5
    Console.WriteLine("x is " & x)
    RefSub(x)
    Console.WriteLine("x is " & x)
End Sub

Sub RefSub(ByRef x As Integer)
    x = 10
    Console.WriteLine("x is " & x)
End Sub
```

Any Difference in Output?

Local Variables

- Variables declared inside a Sub procedure with a Dim statement
- Space reserved in memory for that variable until the End Sub – then the variable ceases to exist

Local Variable Example

```
Sub LocalTester()
   TestLocals()
   TestLocals()
End Sub

Sub TestLocals()
   Dim I As Double
   Console.WriteLine("I is " & I)
   I = 10
   Console.WriteLine("I is " & I)
End Sub
```

Output:
I is 0
I is 10
I is 0
I is 10

Class-Level Variables

- Visible to every procedure in a form's code without being passed
- Dim statements for Class-Level variables are placed
  - Outside all procedures
  - At the top of the program region
  - Useful for variables you would like to use within many procedures on the form

Class Level Example

```
Public Class Form1
   Inherits System.Windows.Forms.Form

   Dim strName As String

   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
```

Scope

- Class-level variables have class-level scope and are available to all procedures in the class
- Variables declared inside a procedure have local scope and are only available to the procedure in which they are declared

Debugging

- Programs with Sub procedures are easier to debug
- Each Sub procedure can be checked individually before being placed into the program
- A little later we will see how to use the built-in debugging tool

In-Class Exercise

- Write a subroutine that swaps two integer variables; e.g. Swap(x,y) results in exchanging the values in X and Y

4.3 Function Procedures

- User-Defined Functions Having Several Parameters
- Comparing Function Procedures with Sub Procedures
- Collapsing a Procedure with a Region Directive

User-Defined Functions

- Similar to a Sub Procedure, but Functions always return **one** value
- Syntax:

```
Function FunctionName(ByVal var1 As Type1, _
                      ByVal var2 As Type2, _
                        …) As dataType
  statement(s)
  Return expression
End Function
```

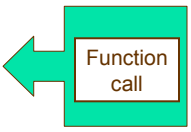

Some Built-In Functions

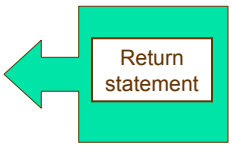
Function	Example	Input	Output
Int	Int(2.6) is 2	number	number
Chr	Chr(65) is “A”	number	string
Asc	Asc(“Apple”) is 65	string	number
FormatNumber	FormatNumber(12 345.628, 1) is 12,345.6	number, number	string

Sample

```
Private Sub btnDetermine_Click(...)
    Handles btnDetermine.Click
  Dim name As String
  name = txtFullName.Text
  txtFirstname.Text = FirstName(name)
End Sub

Function FirstName(ByVal name As String) As String
  Dim firstSpace As Integer
  firstSpace = name.IndexOf(" ")
  Return name.Substring(0, firstSpace)
End Function
```





Having Several Parameters

```
Private Sub btnCalculate_Click(...)
    Handles btnCalculate.Click
  Dim a, b As Double
  a = CDbl(txtSideOne.Text)
  b = CDbl(txtSideTwo.Text)
  txtHyp.Text = CStr(Hypotenuse(a, b))
End Sub

Function Hypotenuse(ByVal a As Double, _
                    ByVal b As Double) As Double
  Return Math.Sqrt(a ^ 2 + b ^ 2)
End Function
```

User-Defined Functions Having No Parameters

```
Private Sub btnDisplay_Click(...) _
    Handles btnDisplay.Click
  txtBox.Text = Saying()
End Sub

Function Saying() As String
  Return InputBox("What is your" _
          & " favorite saying?")
End Function
```


Comparing Function Procedures with Sub Procedures

- Subs are accessed using a call statement
- Functions are called where you would expect to find a literal or expression
- For example:
  - Result = *functionCall*
  - lstBox.Items.Add (*functionCall*)

Functions vs. Procedures

- Both can perform similar tasks
- Both can call other subs and functions
- Use a function when you want to return one and only one value
  - A function can also be declared with ByRef arguments to return multiple values back through the argument list

Collapsing a Procedure with a Region Directive

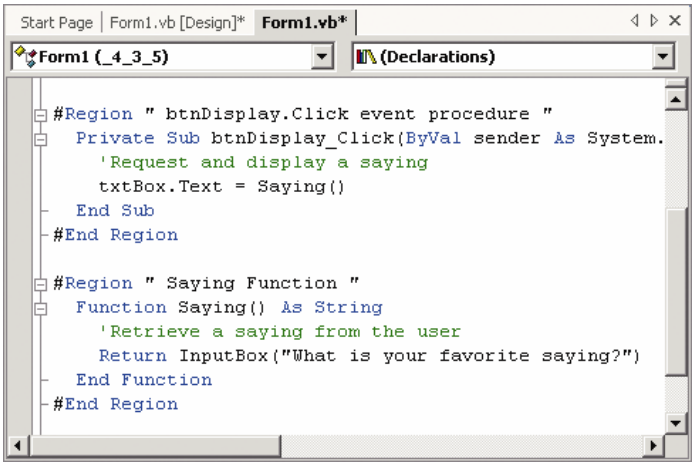
- A procedure can be collapsed behind a captioned rectangle
- This task is carried out with a **Region directive**.
- To specify a region, precede the code to be collapsed with a line of the form

```
#Region "Text to be displayed in the box."
```

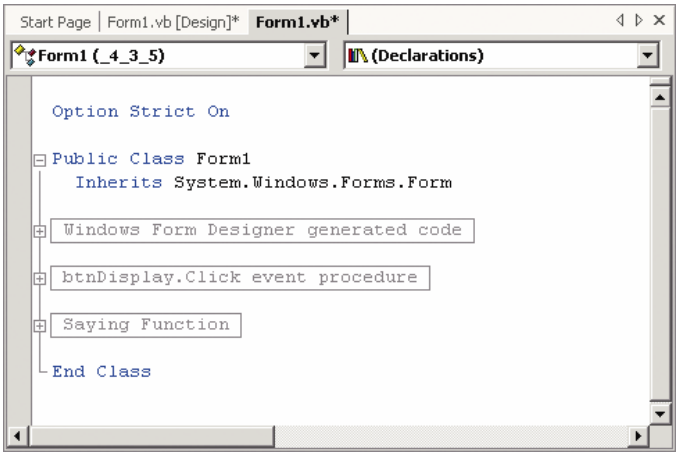
- and follow the code with the line

```
#End Region
```

Region Directives



Collapsed Regions



In-Class Exercise

- The following example shows how to generate a random number: (we'll explain the **new** later)
  - Dim r As New Random()
  - r.Next(intMin, intMax)
    - Returns a random number ≥ intMin, < intMax
- Write a function named RollDice that simulates rolling two six-sided dice and returns the sum of the roll
  - Print out several rolls to see if it is working

4.4 Modular Design

- Top-Down Design
- Structured Programming
- Advantages of Structured Programming

Design Terminology

- Large programs can be broken down into smaller problems
- "divide-and-conquer" approach called "stepwise refinement"
- Stepwise refinement is part of top-down design methodology

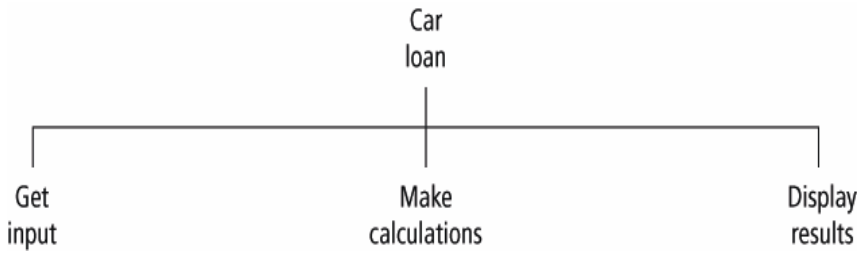
Top-Down Design

- General problems are at the top of the design
- Specific tasks are near the end of the design
- Top-down design and structured programming are techniques to enhance programmers' productivity

Top-Down Design Criteria

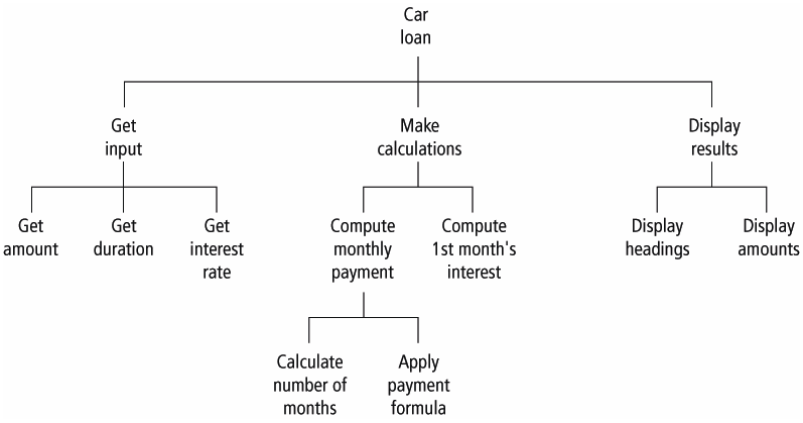
1. The design should be easily readable and emphasize small module size.
2. Modules proceed from general to specific as you read down the chart.
3. The modules, as much as possible, should be single minded. That is, they should only perform a single well-defined task.
4. Modules should be as independent of each other as possible, and any relationships among modules should be specified.

Top-Level Design HIPO Chart



HIPO = Hierarchical Input Process Output

Detailed HIPO Chart



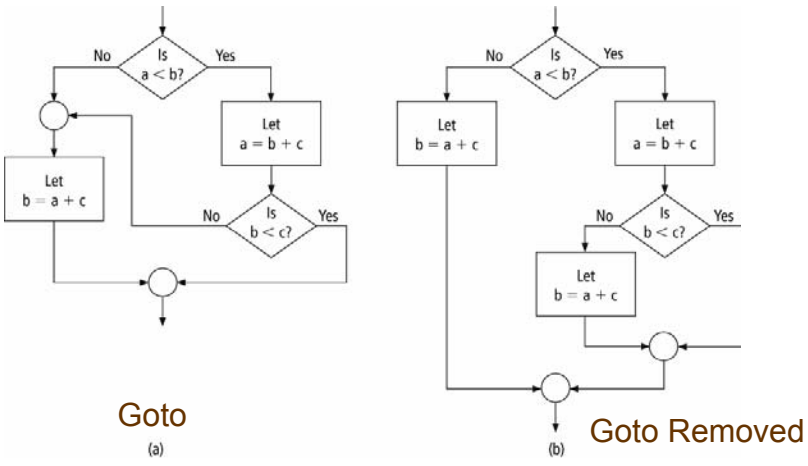
Structured Programming

- Control structures in structured programming:
- ***Sequences:*** Statements are executed one after another.
- ***Decisions:*** One of two blocks of program code is executed based on a test for some condition.
- ***Loops (iteration):*** One or more statements are executed repeatedly as long as a specified condition is true.

Advantages of Structured Programming

- Goal to create correct programs that are easier to
  - write
  - understand
  - modify
- "GOTO –less" programming

Comparison of Flow Charts



Easy to Write

- Allows programmer to first focus on the big picture and take care of the details later
- Several programmers can work on the same program at the same time
- Code that can be used in many programs is said to be reusable

Easy to Debug

- Procedures can be checked individually
- A driver program can be set up to test modules individually before the complete program is ready
- Using a driver program to test modules (or stubs) is known as stub testing

Easy to Understand

- Interconnections of the procedures reveal the modular design of the program.
- The meaningful procedure names, along with relevant comments, identify the tasks performed by the modules.
- The meaningful variable names help the programmer to recall the purpose of each variable.

Easy to Change

- Because a structured program is self-documenting, it can easily be deciphered by another programmer (at least, easier than if it was unstructured!)

Object-Oriented Programming

- an encapsulation of data and code that operates on the data
- objects have properties, respond to methods, and raise events.

- We will discuss OOP in more detail the last week of class