CMPT 110 Study Guide
Unit 5: Subprograms

Adapted from Tony Dixon’s Distance Education Version of CMPT 110 at SFU

Introduction

Objectives

- To learn how to call and use the built-in functions of Visual Basic.
- To learn how to apply methods to objects and to become familiar with some methods associated with string objects.
- To learn how to write your own "user-defined" Sub procedures.
- To understand the concept of top-down programming and how hierarchy charts facilitate this programming style.
- To understand why parameter passing is preferred over shared variables when information must be exchanged between subprograms.

Subprograms

Productive programming is about reusability. This means that each time we need to write some source code to perform a particular task, we should make use of any previously written programs rather than "reinvent the wheel." Another opportunity for taking advantage of reusability occurs when we anticipate that we will be performing the same calculation or task repeatedly in our program or perhaps using the same calculation in several programs. Both of these opportunities occur continually in writing programs, and for this reason the topic of subprograms is central to the practice of effective programming.

A subprogram is a group of programming statements that are "self-contained," which means that there is limited access to the workings of those programming statements. A subprogram is evaluated whenever it is necessary to perform the calculation expressed by the sequence of statements that make up its body. The only communication that is permitted with a subprogram is the transfer of any values necessary to perform the calculation and the transfer of any result that is produced.
Previous examples of subprograms include the control methods that need to be defined in order to respond to user interface events in a window. Other examples are the built-in functions that permit the programmer to perform familiar calculations such as the trigonometric functions or the string functions.

There are three types of subprograms:

1. A "function" is the term used to describe a subprogram whose purpose is to compute a value. For example, "Sqrt" is the name of a function to compute the square root of a numeric value.
   NOTE: The textbook uses the term "Function procedure" when referring to functions.
2. A "procedure" is the term used to describe a subprogram whose purpose is to cause an effect to occur. An example is an output procedure where the effect is to display a value on a monitor or write a value to a file.
   NOTE: The textbook uses the term "Sub Procedure" when referring to procedures.
3. A "method" is the term used to describe a subprogram whose purpose is to modify an object. Some examples of methods were introduced in the previous unit. Since modifying an object is an "effect", methods are similar to procedures.

Subprograms that are provided as part of the Visual Basic software library are called "built-in." The VB library provides built-in functions, built-in procedures and built-in methods.

Subprograms that are defined by a programmer are called "general." Programmers can write their own general functions, general procedures, or general methods.

**System-Defined Functions**

The Visual Basic development environment (VS IDE) provides a large number of predefined (also called "built-in" or "system-defined") functions that are capable of providing a wide variety of frequently used calculations. Such functions relieve you of the need to write programs that perform these calculations yourself. For example, if you need trigonometric functions such as SIN, COS, or TAN, you can make use of an existing "system" function that was previously written to perform these tasks.

So, how do you use such functions in your programs? You do so in much the same way as you would express the need for such a function in a mathematical expression: by naming it and supplying a value to which the function is to be applied. For example, mathematically we might express the sine of an angle \( x+\pi \) as:

\[
\sin(x+\pi)
\]

with the understanding that that "\( \sin \)" refers to the trigonometric function "sine." In Visual Basic we can represent the same expression as:

\[
\text{Sin}(x + \text{pi})
\]
"Sin" is the name of the Visual Basic Function subprogram to compute the sine of an angle and "pi" is a Visual Basic Constant that has been previously defined to be the literal 3.1415926. The expression inside the parentheses is called the "argument" of the function.

In mathematics, we may wish to evaluate the expression or simply use it to represent the value symbolically. In Visual Basic, however, every such expression will be evaluated if it gets executed. If we wish to save the value, we can do so with an assignment statement:

```vbnet
result = Sin(x + pi)
```

Alternatively, we may wish to include it as part of a larger calculation:

```vbnet
answer = 2. * a * Sin(x + pi) - b * Cos(y - pi/2.)
```

In the previous unit, you were encouraged not to allow Visual Basic to decide what the data type of a result should be. Since the values returned by the Sin and Cos VB functions are real numbers, the preferred data type of the variables answer, x and y, the constant pi, and the literal 2 should be represented by data type Double. Hence the decision to represent the value 2 by a Double data type literal rather than an Integer data type literal.

If you have previously included the statement:

```vbnet
Option Strict On
```

then the choice of representation of the literals to be of data type Double is required.

Of course, there are occasions when you want the value of the expression to be integer. But how are you to do it?

The answer is that Visual Basic provides you with built-functions to use when conversion is necessary. The following three are particularly useful for the data types that are discussed in this Study Guide:

- `CDbl( <value>)` converts the argument `<value>` to data type Double
- `CInt( <value>)` converts the argument `<value>` to data type Integer
- `CStr( <value>)` converts the argument `<value>` to data type String

When you provide input to your program through a TextBox, data entered always has data type String. Therefore, if you wish to use numeric data entered through a TextBox, your program must convert the value in the TextBox to either the Double or the Integer type using one of the appropriate functions given.

Similarly, Textboxes can be used to display the results of calculations. But since the result of a calculation is numeric, you must first convert it to a String before assigning it to the Text property of a TextBox.
Visual Basic provides many other useful functions that save you time as a programmer since they mean you don't have to write your own solutions. Therefore, you should familiarize yourself with what functions are available to you.

VB functions are grouped in packages called "modules." The VS IDE can be used to find out more information about a module or function if you know its name. For example:

1. From the View menu, select Object Browser. Click on it.
2. Underneath the word "Browse:" is a large TextBox used for searching for a specific name. Enter the name "Financial" And press Enter. In the right half of the Object Browser window you will see displayed the name and parameter list of a number of functions for performing financial calculations.
3. Scroll down and click on the function named "SLN." At the bottom of the window you will see information displayed about the function - its declaration statement, a brief statement of purpose, and the role of each of its formal parameters.

By specifying different modules and then clicking on the names of different functions, you can find information about their syntax and purpose.

System-Defined Methods

The term "method" is used to refer to a procedure that acts on instances of a class of objects. Some of the objects you have encountered so far are control objects - for example, TextBoxes and Buttons. A String is also an example of an object. Whenever you use a method, you have to precede the method's name by the name of the object for which the method is applicable followed by a period.

A "built-in method" is one that is already provided for you as part of the Visual Basic software to use in your programs. Some built-in methods are:

<table>
<thead>
<tr>
<th>METHOD</th>
<th>EFFECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Me.Close</td>
<td>In this case the Close method is being applied to the object Me, whose value is the Form that is currently being displayed.</td>
</tr>
<tr>
<td>TextBox1.GotFocus(sender, e)</td>
<td>The GotFocus method is being applied to the object TextBox1.</td>
</tr>
<tr>
<td>&lt;String&gt;.ToUpper</td>
<td>The ToUpper method creates a new string whose characters are the upper case versions of the letters in the original object, &lt;String&gt;.</td>
</tr>
<tr>
<td>&lt;String&gt;.ToLower</td>
<td>The ToLower method creates a new string whose characters are the lower case versions of the letters in the original object, &lt;String&gt;.</td>
</tr>
</tbody>
</table>
SOME BUILT-IN METHODS

<table>
<thead>
<tr>
<th>METHOD</th>
<th>EFFECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;String&gt;.Trim</td>
<td>The Trim method creates a string with all leading and trailing &quot;space&quot; characters removed.</td>
</tr>
<tr>
<td>&lt;String&gt;.IndexOf(&lt;substring&gt;)</td>
<td>The IndexOf method determines the starting position of the first occurrence of the string given by &lt;substring&gt; in the object &lt;String&gt;.</td>
</tr>
<tr>
<td>&lt;String&gt;.Substring(&lt;start&gt;,&lt;length&gt;)</td>
<td>The SubString method determines the substring of the object &lt;String&gt; that begins at position &lt;start&gt; and is &lt;length&gt; characters in length.</td>
</tr>
</tbody>
</table>

It is important to know that the left-most position of a character in a string is position 0 (and not position 1). Therefore, the expression:

"CMPT 110".IndexOf("110")

evaluates to the number 5 since "110" begins in position 5.

Similarly,

"CMPT 110".Substring(4,3)

evaluates to "11" and not to "T 1".

Note that the space between "T" and "1" counts as a character, the "blank" or "space" character. In the string "CMPT 110" it is located in position 4 and therefore 4 would be the value of the expression:

"CMPT 110".IndexOf(" ")

There is another important fact to remember about strings: Do not confuse the string "" (no spaces between the quotation marks) with the string " " (one space between the quotation marks). The first string is a string of length 0 called the "null string," and the second is a string of length 1 consisting of the single character "blank."

User-Defined Methods

You have already seen examples of how Sub procedures are defined when you created the bodies of methods for responding to windows events. Like these methods, every Sub procedure defines a block of programming statements, beginning with a declaration statement that provides the name of the Sub procedure and a list of variables, called "formal parameters." The formal parameters provide the means through which values can be exchanged between the Sub procedure and the program that is using the Sub, which is referred to as the "calling program."
Like the methods you have seen, following the declaration statement, the Sub procedure consists of a sequence of programming statements that collectively form the "body" of the Sub procedure. The Sub procedure block concludes with an "End Sub" statement, just like a method.

The simplest Sub procedures have no formal parameters. Everything needed to perform the task for which they are written is contained within the body or provided through shared variables.

You might ask, "What kind of task can be described in this manner?" Primarily, such tasks are those that always do exactly the same thing whenever required with no variation in what actions are taken. A good example is a subprogram to produce the same heading on each page of a report, perhaps the name of a company or the title of a report.

As an example, the following Sub procedure will clear the contents of four TextBoxes that presumably are part of a Form:

```
Sub ClearTextBoxes()
    TextBox1.Text = ""
    TextBox2.Text = ""
    TextBox3.Text = ""
    TextBox4.Text = ""
End Sub
```

Why not simply type in the four statements that actually clear the TextBoxes every time you wish to clear them? The answer, of course, is that with this Sub procedure you only need to type either of the following statements:

```
Call ClearTextBoxes()
```
or, more simply:

```
ClearTextBoxes()
```

each time you wish to clear all the boxes. Since you may wish to do so in a variety of different places in your program, you are saving three lines of code each time. An additional benefit is that the name of the Sub procedure can provide information about what is happening without showing how. The benefit of this fact is probably not immediately apparent. However, suppose you are only given that part of the program that calls the Sub procedure. Then you will only know that a Sub procedure named "ClearTextBoxes" is being called, and from its name, you know what it is doing but not how. If the name accurately describes the behaviour and the Sub procedure performs
its task correctly, then it should not matter to you how the Sub procedure actually does so. This is analogous to the description inside a box of a hierarchy chart.

Procedural Abstraction

The use of a procedure name to describe a task without identifying the details of how the task will be performed is called "procedural abstraction." It is an important programming technique and can influence the way in which you go about the task of writing a program. For example, suppose you wish to write a program that clears all the TextBoxes and draws an x-axis and a y-axis in a PictureBox when it begins execution. Suppose, further, that you are not quite certain yet how many TextBoxes you will have on your Form. Also, unless you have read ahead, you may be uncertain about what PictureBoxes are. Despite this missing information, you can still make a start on your program by employing procedural abstraction.

Suppose a partial hierarchy chart for a Graphing program is:

If these tasks are to be performed upon the click of a "START" button, Then you should conclude that you will need to perform the necessary tasks inside a Button_Click method. Such a method might be defined as follows:

```csharp
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles MyBase.Load
    ClearTextBoxes ()
    DrawGraphAxes ()
    PlotGraph ()
End Sub
```

The statements "ClearTextboxes", "DrawGraphAxes", and "PlotGraph" are referred to as "calling statements to Sub procedures" or, more simply, "Sub calls." Although the actual Sub procedures will need to be defined before the program can be run, the calling statements to suitably named procedures describing their purpose can be defined prior
to actually writing the Subs themselves! Note how the body of the Button1_Click method describes what needs to be done not too differently from the original hierarchy diagram. That is, the method will "clear some textboxes, then draw some graph axes," and finally plot the graph.

We can even compile the source code successfully (i.e., without error messages) by providing preliminary Sub procedures that do nothing if executed, since they have no bodies:

```vbnet
Sub ClearTextBoxes()
End Sub

Sub DrawGraphAxes()
End Sub

Sub PlotGraph()
End Sub
```

These Subs are called "procedure stubs" because they do not represent the final versions but only permit the program to be compiled. Without them, an error message will occur indicating that ClearTextBoxes, DrawGraphAxes, and PlotGraph are undefined (try it!).

Once you have determined how many TextBoxes you will employ, what a PictureBox is, and how to draw an x-axis and a y-axis, then you will be able to "fill in" the details about how the ClearTextBoxes, DrawGraphAxes, and PlotGraph Subs will perform their tasks.

This approach is called "top-down" programming. The basic strategy is to define a program as a sequence of major tasks that need to be performed as you would do in a hierarchy chart. By giving a name to each task (represented by a box in the hierarchy chart), the programmer identifies the need for a procedure that will also need to be written at a "later time" when the details of how to achieve each task are determined. Until these details are determined, each of the Sub procedures is represented by a procedure stub.

The programming of large projects can be accomplished very effectively using this approach. Rather than puzzle about "where to start" when tackling a large project, the programmer begins by breaking the problem down into a series of simpler programming problems until each piece can be addressed directly. A hierarchy chart is an effective way to begin this process. As a solution to each piece is found, its procedure stub is replaced by a version of the procedure that defines how to actually perform the specified task.
The Procedure Interface

The previous examples of Sub procedures did not require a means to exchange information. However, it is more often the case that a Sub procedure will require values in order to perform its task. For example, the "DrawGraphAxes" Sub suggested previously presumably will always draw the axes in the same place in a PictureBox and to the same scale unless we provide information on where to place the axes and what scale to use. There are two ways that we can provide such information to the procedure:

1. **Shared Variables.** With this approach, variables that are declared outside the Sub procedure block are assigned values in the programming block that includes the calling statement, which, of course, is also outside the Sub procedure block.

2. **Parameter Binding.** In this case, a list of variables, called "formal parameters," is defined in the declaration statement of the Sub procedure. When a Sub is called, values are assigned to these parameter variables by including in the calling statement a list of values, called "arguments," that are to be assigned to the parameter variables. The arguments are provided in the calling statement in the same order as the variables in the parameter list of the Sub procedure declaration statement.

Parameter Binding is preferred over Shared Variables when passing values to Sub procedures for two reasons. First, the information about the data type of a parameter is part of the declaration statement of the Sub, and so it is readily located by anyone looking at the Sub who wants to know this information. The location of a shared variable is sometimes not easily identified, and so those seeking its data type may find themselves looking through the entire program.

A more important concern however is that shared variables are vulnerable to unintentional change. By their nature, almost any part of the program can access them and change their values. Parameter variables, on the other hand, can only be changed within the body of the Sub procedure where they are declared.

A common occurrence in programming is for the programmer to use the same variable name more than once in a program. This can result in undesirable results if such variables are shared. However, the same name can be used in the parameter list of every general procedure if the programmer so desires without any undesirable effects since in each case the value of the parameter is confined to the body of the Sub in which it is declared. You may already have observed this reuse of a variable name in the method declarations of the windows event methods for control objects such as Buttons and TextBoxes.

These observations lead to another common programming convention. Always use parameter binding rather than shared variables to pass values to general procedures except in very limited circumstances. Further, if shared variables must be used, keep the number to an absolute minimum.
The following example illustrates the syntax for declaring parameters and for providing arguments in calling statements. Suppose we wish to provide information to the DrawGraphAxes Sub procedure so that the programmer can specify the scale and location of the x-axis and y-axis. One way to do so is to provide the minimum and maximum x-co-ordinate and y-co-ordinate values. Four parameter variables are therefore required, and we can name them xmin, xmax, and ymin, and ymax. Assuming that the values assigned to these parameter variables will be real numbers, the declaration statement for the Sub procedure is:

```
Sub DrawGraphAxes (ByVal xmin As Double, _
    ByVal xmax As Double, _
    ByVal ymin As Double, _
    ByVal ymax As Double)
```

The calling statement to DrawGraphAxes must now provide arguments for each of these four parameters, and their values must be provided in the same order as the parameters appear in the declaration. Suppose we wish the x-axis to run from -50 to +70 and the y-axis to run from -25 to +75. Then the appropriate calling statement is:

```c
DrawGraphAxes(-50.0, 70.0, -25.0, 75.0)
```

where -50.0, 70.0, -25.0, and 75.0 are the argument values to be assigned to the formal parameters xmin, xmax, ymin, and ymax.

In Unit 11 we will revisit this example by writing the body of a related Sub procedure, DrawGraphAxes, where we will use a different method for passing the values to the procedure.