Procedures

A **procedure** is a block of code that performs a specific task. In VB .NET, there are 2 types of procedures – event procedures and general procedures.

An **event procedure** is a block of code that runs when an event occurs, like a button click.

A **general procedure** is a programmer-defined block of code that simplifies a program by dividing it into smaller, more manageable chunks. General procedures can reduce redundancy in the code by performing a task that may need to be executed several times throughout the program – they are reusable, written once and used many times. It's much easier to debug a program that is in smaller modules of code. It's must faster to develop a project when multiple programmers can work on different procedures at the same time. Some procedures can be re-used outside of the original project, which can save many hours of coding and testing. For example, the InputBox and MessageBox procedures are available in any VB .NET program – these general procedures were written and tested thoroughly and included for anyone to use.

Each general procedure has a name and specific job which should be very focused and defined. A general procedure is a specialist, doing one thing very well. A washing machine can wash clothes but not dishes – it specializes in washing clothes. If you put dishes in a washing machine, you won't have many dishes left!

Another example is the drive-through at your favorite fast food restaurant. You give your order to the clerk to make a hamburger. Does that clerk go to the grill and make the hamburger? Usually there is another worker who does the cooking, or performs that procedure. The clerk "calls" or "invokes" the procedure to cook the hamburger; another worker does that job, and "returns" the burger to the clerk, who has to wait for that cooking procedure to be completed before handing the bag with the food to you.

There are two types of general procedures – **subs** and **functions**. Subs do a job and return, and functions do a job and return with a new piece of data. That washing machine might be considered a sub – it does its job and stops running. A calculator might be considered a function – you enter numbers and an operator, and it gives you an answer, a new piece of data. The MessageBox procedure is a sub – it does its job of delivering some message to the user, then ends and control returns to the main program. The InputBox procedure is a function – it takes the data the user provides and delivers it to the program.

One big difference between event procedures (such as click events) and general procedures (subs and functions) is how they start running. When you click on a button on a form, the procedure or code for that click starts executing – the procedure responds to an event.

General procedures do not begin execution because of an action by the user or an event on the form. They are **called** from the code; that is, they begin running because some line of code told them to run. General procedures can be considered "code within the code". As the program runs, it executes one line of code, then the next line, and so on. When it gets to a call to a general procedure, it puts a bookmark in that location, looks through the file to find the code for the procedure, then switches to the

procedure to run it. The calling program gets swapped out of memory, the called procedure moves into memory and executes; when the called procedure ends, it swaps out of memory and the calling program returns to memory and picks up exactly where it left off. This corresponds to reading your textbook, putting a bookmark in a certain spot, flipping to the back of the book to look something up in the index or an appendix, then returning to the bookmark to continue reading where you left off.

A block of code can call as many different procedures as it takes to accomplish the task. It may also call the same procedure multiple times, if that's what it takes to solve the problem. A procedure may call another procedure as well. They are very versatile and powerful tools in your programming back of tricks.

Every general procedure needs to be defined, just as every variable needs to be declared. Procedures cannot be defined inside another procedure, so they are written outside of all procedures. Every project looks different but every program has "End Class" as its last line, so it's safe to say you should go to that last line and type procedures directly above it.

Defining a Sub: The format for defining a sub looks like this:

```
Private Sub SubName(parameters)
        statements to perform some work
End Sub
```

You can write as many lines of code as necessary to perform the work. Remember that a general procedure should be focused and specific in its task. Don't make it overly complex by trying to do too many things in one procedure.

Look at the following example, a sub that will clear or reset two variables back to zero:

```
Private Sub ResetStuff()
        Counter = 0
        Amount = 0
End Sub
```

Calling a sub requires simply saying its name on a separate line of code. It may begin with the keyword "Call", but that is not required. These two examples are equivalent:

```
Call ResetStuff( )

ResetStuff( )
```

Defining a Function: The format for defining a function is as follows:

```
Private Function FunctionName(parameters) as ReturnType
        statements to perform some work
        Return something
        OR
        FunctionName = something
End Function
```

The function returns a single value to the calling program, and the data type of that new piece of data must be specified – that's the ReturnType at the end of the first line above. You return that data either by assigning the value to the name of the function or with the keyword **Return**. Both let the compiler know what value will be returned to the calling program when the function ends. You can put as much code as you need in the function to figure out the value to return. It can be very simple, like a single line to generate a random number, or very complex, with many lines of code.

Look at the following example, a function that will return a string containing "Rock", "Paper" or "Scissors":

```
Function GetRPS( ) as String
    Select Case Int(RND ( ) * 3 + 1)    'generate a random number
    Case 1
        GetRPS = "Rock"
    Case 2
        Return "Paper"
    Case Else
        Return "Scissors"
    End Select
End Function
```

Notice that in case 1, the new data was returned to the calling program by assigning the data to the name of the function. In cases 2 and 3, the keyword "return" was used to send the new data back to the calling program. Both work the same, it's a matter of personal or institutional preference about which way to return data from a function.

Functions always return a new piece of data, and you need to use that data in the calling program. You either use an assignment statement to catch that data and store it, or use a test to check that data. The function IsNumeric tests if some data is a number and returns True or False as the new piece of data. You can test that new piece of data:

```
If IsNumeric(text) then
```

Or you can store that new piece of data in a variable:

```
goodNumber = IsNumeric(text)
```

You **must** catch the data being returned from a function, either in a variable or in a test. Therefore, calling a function always happens in an assignment statement or a test like IF or Case.

PARAMETERS

Every general procedure has parentheses after its name, which is the place to list the data the procedure needs to do its job. This parameter list may be empty, as in the GetRPS( ) example; it may have one variable, as in the IsNumeric(variable) example; or it may have more variables as needed. Remember that every general procedure must have parentheses – that's one of the clues that it's a procedure and not a variable.

What data does the procedure need to do its job? That is determined when the procedure is defined, and those requirements are listed inside the parentheses, in the parameter list. There can be as many parameters as needed for the procedure to complete its work. The parameters can be of any data type, and different parameters can be different data types. For example, if a procedure calculates the area of a rectangle, it needs length and width of that rectangle, and both are numbers. If a procedure posts the test score for a student, it needs the student name and the test score, a string and a number. If a procedure prints a name, it needs that name, a string.

When VB .NET passes data to a procedure, it can pass a copy of the variable in the calling program, or the actual variable. If it passes a copy (the default behavior), the original variable is protected, cannot be changed by the procedure. If it passes the actual variable, it sends the address reference for the variable, and the procedure can change the variable permanently. This can be very helpful, but also could be very damaging, so it's important you understand this clearly.

The format for a single parameter in a procedure definition requires 3 pieces of data – how it's being passed (copy or the real thing), its name in the procedure, and its data type. You must provide all 3 pieces of data for every parameter in the list.

Passing copies of variables uses the keyword "ByVal". This data is being passed by value, a copy of the data. Passing the real variable uses the keyword "ByRef", sending the address reference for the variable. The format for defining parameters look like this:

Passing by value (copy of the data)	Passing by reference (the actual variable)
ByVal parameterName as DataType	ByRef parameterName as DataType

Let's look at some examples. In the function that calculates the area of a rectangle, the function definition may look like this:

```
Private Function CalculateArea( ByVal decLength as Decimal, ByVal decWidth as Decimal) as
Decimal
    Dim decArea as Decimal
    decArea = decLength * decWidth
    Return decArea
End Function
```

In the sub that writes a student's test score in a label, the definition may look like this:

```
Private Sub DisplayScore(ByVal strName as String, ByVal decScore as Decimal)
    lblResult.Text = "Student Name: " & strName & ", Score: " & decScore
End Sub
```

Consider an example where the score for a game needs to be updated. This procedure does not need to create a new piece of data, it needs to change the value of an existing piece of data.

```
Private Sub UpdateScore(ByRef decPoints as Decimal)
    decPoints += 100
End Sub
```

Because the parameter was passed ByRef, when its value changes in the sub, it changes permanently.

Consider an example where the score needs to be updated based on the risk the user chose.

```
Private Sub FixScore(ByRef decPoints as Decimal, ByVal decRisk as Decimal)
    decPoints = decPoints + (2 * decRisk)
End Sub
```

In this example, the decPoints variable needs to be changed permanently, so it's passed ByRef. The decRisk variable can be a copy of the data because it won't change, it's just used in the calculations, so it's passed ByVal.

Each parameter is separate and unique, so each one must be fully defined about how it is passed, its name, and its data type. To add another parameter, add a comma and provide those 3 pieces of data in the parameter list.

In all this discussion of parameters, we've been looking at the procedure definition, with no thought of the calling program. Remember that one programmer might be writing a procedure and a different programmer writing the program that calls the procedure. They connect through the parameter list. Programmer P, writing the procedure, will assign names to all the variables in the procedure. Separately, Programmer M, writing the main program that calls the procedure, assigns names to all the variables in the main program. Those variable names do not have to match – that's one of the greatest strengths of general procedures, and makes them reusable in other programs. What does have to match is the parameter list – the number of variables, the data types of the variables, and the order of the variables listed in the parameter list.

Consider the parameters as parking places for data. When calling a procedure, the data passed to it is an "argument", the actual data used for that specific call. Consider arguments as the automobile being parked in the parking places. Get it? Parameters are parking places, arguments are autos. The parking places are defined when the procedure is defined; when that procedure is called, the autos go in the parking places.

In the example for printing student name and score, the parameter list specifies a string for the name and a decimal for the score. So the calling program must provide those same data types.

```
Definition:  Private Sub DisplayScore(ByVal strName as String , ByVal decScore as Decimal )
      Call:          DisplayScore(      strStudent      ,      decTestScore      )
```

Notice the names of the variables are different in the call than the definition, but the number of variables and the data type of those variables match. The program would not compile if the call were written like this:

```
DisplayScore(decTestScore, strStudent)
```

The compiler matches the variables by position and data type – the data type of the first variable in the call must match the data type of the first variable in the definition of the parameter list of the procedure, and so on.

Definition:	Function	GetPay(ByVal Hours as Double,		ByVal Rate as Double)		as Double
Call:	dblTotalPay =	GetPay(	20.0	,	10.50	)
Call:	dblPay =	GetPay(	dblHours	,	dblRate	)
Call:	OvertimePay =	GetPay(	dblHours – 40	,	dblRate * 1.5	)

Note that the keywords ByVal and ByRef do not appear in the CALL to the procedure, only in the definition. Also the data types are not specified in the CALL, only in the definition.

SUMMARY

Procedures: blocks of code

Event procedures: procedures triggered by an event, like a button click

General procedures: procedures triggered by a call from code

Functions: general procedures that return a new piece of data

Subs: general procedures that do not return a new piece of data

Definition of a procedure: specifying the parameter list and the work done in the procedure

Calling a procedure: the line of code that transfers control to the procedure

Returning from a procedure: when the procedure ends, control transfers back to the calling code

Parameters: the list of variables a procedure needs to do its job

Arguments: the actual data sent to a procedure in the parameter list

Passing data: the sharing of data from the calling program to the procedure using parameters

Format for defining a sub:

```
Private Sub SubName(parameters)
        ... code to accomplish the task ...
End Sub
```

Format for defining a function:

```
Private Function FunctionName(parameters) As ReturnDataType
        ... code to accomplish the task ...
End Function
```

Format for defining a parameter in the parameter list:

ByVal varName as Datatype OR ByRef varName as DataType

Format for calling a sub:

SubName(arguments) OR Call SubName(arguments)

Format for calling a function:

Assignment statement: someVar = FunctionName(arguments)

Testing: IF Functionname(arguments) = true then