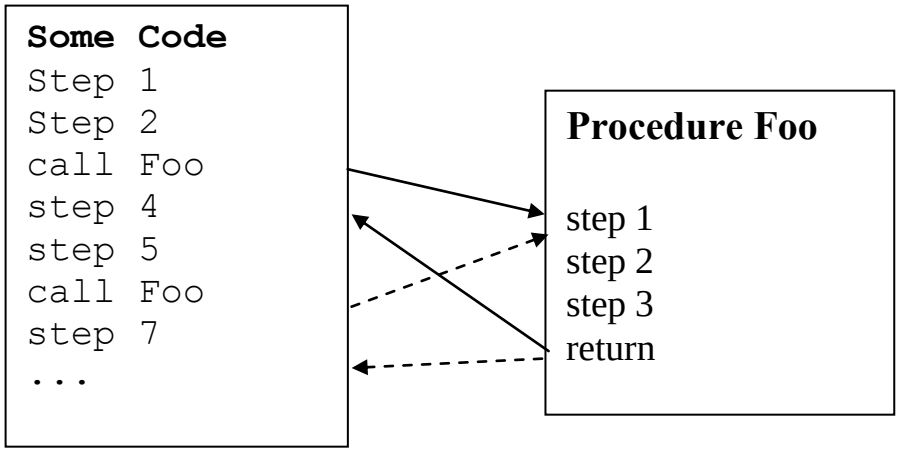
Creating and Invoking Procedures

We've used procedures since our first VB.net programs. We've even let the Visual Studio Designer create event handling procedures for us. You need to know what procedures are, the different types available in VB.net and when to create and use them.

A *procedure* is a name given a sequence of instructions. You can then cause those statements to run by just using the name. Normally the *flow of control* is sequential through our code. When you *invoke* (or *call*) a procedure, all the steps in the procedure get done, and then the sequence picks up from where it left off:



When a procedure is invoked (with a procedure call statement) the flow of control passes to the first statement in the procedure. When the end of the procedure is reached the flow of control then returns to the statement following the procedure call. You can invoke a procedure from several places.

VB.net comes with many procedures you can use. For example you can use `MessageBox.Show` from different places in your programs. You don't need to worry about how this Show method accomplishes its task, just that it does whenever you *invoke* (or *call*) it by name.

> In addition to being invoked programmatically (by a procedure call), some procedures are invoked automatically when an *event* occurs, such as a user clicking on a button in a window. Such procedures are called event handling procedures, or simply event handlers.

Besides the many built-in procedures you can just use, you can create your own. Once created you can use them the same way as the built-in procedures; you can forget about the details of the code inside the procedure, and just use it by name.

Using procedures can help organize your code. For example it is not uncommon to have to perform the same series of steps (a "sub-task") several times in a program. Instead of using copy and paste you can write the code once, and then invoke it several times in your programs. This is a form of ***code reuse*** and is a very powerful technique. Note any bugs fixed in the procedure get used every time the procedure is called. If you used copy-and-paste, you'd have to remember to make the same changes in every location where the original code was copied.

Types of Procedures in VB.net

There are two types of procedures: those whose statements *do* something each time the procedure is invoked, and those whose statements *calculate* something each time:

```
MessageBox.Show( "Hello" )
name2 = UCase(name1)
Console.WriteLine("Square root of 81 is " _
                  & Sqrt(81) )
```

The procedures that do a task are called ***sub procedures***. The ones that calculate something are called ***function procedures***. (Event handlers are a type of sub procedure, never function procedures.)

> In VB.net all procedures must be declared inside either a class or a module. Procedures (of either type) declared inside of a class are also called ***methods***.

In the projects we've done so far, we've always had a Class that represents a form. Anytime we added stuff (constants, event handlers, etc.) we did so inside the class. Later you'll see how to have additional classes and modules used in your programs, but for now we'll stick to a single class.

Top-down Design

Using procedures is important for more than code reuse. Sometimes a task is very complex and takes many steps to accomplish. Lengthy sequences of code are hard to read, understand, debug, and change later (there are always changes).

You can organize a complex task into several sub-tasks. This is called ***top-down design*** (or *step-wise refinement*). It is also known as *delegation*. Think of the boss in a company; the boss will delegate tasks to the various company officers and wait for their results before proceeding with the next task. The officers in turn may delegate some of the work to other flunkies.

When designing a program for some large task, you can't keep track of both the "big picture" and all the details too. (Some large programs are over 1,000,000 lines long!) The way to do this is *top-down design*: ignoring the details, decide on the major sub-tasks that need to be done to complete the overall task. At this point you don't worry about the exact statements to do those tasks. Once this design is done you then can get started working on each sub-task, one at a time.

> In truth, many of the tasks can and should be performed at the same time, not one at a time. However doing so is an advanced topic not covered in our course (known as "parallel programming" or "multi-threading").

Each sub-tasks is done with statements in its own (hopefully short) procedure. When writing the code for some sub-task, you need to "forget" the big picture and focus only on the task at hand. When done with one sub-task, work on the next. A "main" procedure then invokes each sub-task procedure, one at a time and in the correct order, in order to complete the overall task.

If some sub-task seems too complex to program directly, it too can be broken down into sub-tasks (sub-sub-tasks?).

> **Ideally a procedure should do a single task.** (See if you can describe its purpose without using the word "and".) It should be short enough to fit on the screen without scrolling, or at least on a single printed page. Note this is a *rule of thumb* only and not a strict requirement.

When the initial design is done, before any coding of procedures begins, the design is documented and each procedure is given a name and very specific requirements. It is common in fact to create a ***skeleton program*** that contains only comments, modules, classes, and empty procedures known as ***stub procedures***. Even after all that, coding may need to wait for a design review, testing code to be written (*scaffold code*), and a production schedule to be developed (e.g., "programmer Mo will complete procedures A, B, and C by the end of the month, while programmer Jo will complete D, E, and F at the same time. Then after a week of testing and code review, Mo will work on procedures G and H, ..."). Only then will a programmer start coding some procedure.

In real life a programmer will also have responsibilities for testing other programmer's code and designs, manning a support desk, finding and implementing bug fixes, documenting the code (and creating or at least reviewing end user documentation), and other tasks.

Object-Oriented Design

Not all programs are easily decomposed into a hierarchy of tasks and sub-tasks. Different design methods can be used for such programs. One important design method is known as *object-oriented design*. This will be discussed later in the course.

Creating and Using Sub Procedures

To create a sub procedure is easy. Just add the statements you want (known as the procedure's *body*) in between these lines:

```
Private Sub ProcedureName ()
     your code goes here
End Sub
```

Be sure to add some comments at the top so the reader knows what the procedure will do when invoked. (Note you could use `Public` or `Friend` instead of `Private`, but when you only have a single class in the whole program it doesn't matter. The rule to follow is "always use `Private` unless you have a good reason not to".)

To invoke a Sub procedure is also easy. From some other procedure:

```
     ProcedureName
Or:  ProcedureName()   ' using the parenthesis, always, is preferred
Or:  Call ProcedureName
Or:  Call ProcedureName()
```

To create and call an event handler is exactly the same, except you add a list of the events the procedure can handle:

```
Private Sub ProcedureName () Handles something
```

Passing Parameters (Arguments) to Procedures

Procedures are great, but as defined so far are limited. Every slightly different tasks needs its own procedure. There's no way to have a more *general purpose* procedure that can do a number of very similar tasks. Or is there?

Consider a simple example of displaying a pop-up window with a message in it. Without any parameters, every message you want to display would need its own procedure! Try to imagine what that would be like:

```
MessageBox.ShowHello()
MessageBox.ShowWorld()
```

Instead we can pass some data into the procedure, so that the statements inside can do the same steps but with the different data, every time the procedure is invoked:

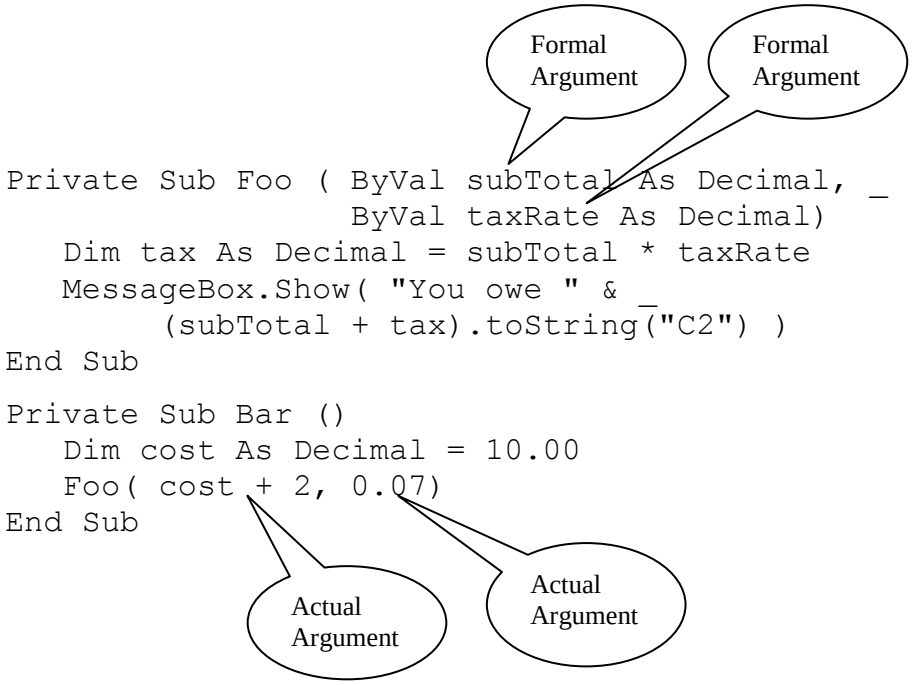
```
MessageBox.Show( "Hello" )
MessageBox.Show( "World!" )
```

Here I invoke the Show procedure twice, each time with a different argument. This is much more flexible!

The data passed to a procedure when you call it, is called a ***parameter*** or ***argument*** (it depends on who wrote your textbook).

But don't make your procedures too general, that is don't have a DoAnything() procedure. Each procedure should perform a single task, such as display a pop-up window with some message in it.

When invoking procedures you can pass in none, one, or more arguments. If you don't pass in any, you don't need the parenthesis after the procedure name if you don't want to use them. (I do, because most other languages require them regardless.) The number of arguments passed must match the number that were declared when you wrote the procedure. Here's an example:



To help distinguish between arguments in a procedure definition and arguments in a procedure call, some people call **the first *formal argument* and the second *actual arguments*.** (Others like the term *argument* only to apply to actual arguments, and use the term *parameter* to refer to formal arguments.)

Each actual argument can be a literal value or an arbitrary expression, even one that involves invoking other functions. It can also be the name of a variable. However all formal arguments are simply the name of the local variable that will hold the value passed in.

Before invoking the procedure all the arguments are evaluated. These are made available to the code by one of two methods (in modern languages anyway):

Pass by copy (sometimes called *pass by value*) is the most commonly used way. Each *actual argument* is *copied* into a new variable, the *formal argument*. This protects the original arguments, since any changes made in the procedure only affect the copy.

Pass by copy works just like assignment. If the types of the actual and formal arguments aren't the same, the actual argument is converted into the correct type and then assigned to the formal argument. This is the default method in VB.net.

Some languages such as C and Java only support pass by copy.

(*Example of how pass by copy works: make a text file on the desktop "foo.txt". Then make a copy of it "bar.txt". Now make a change to bar.txt and show how foo.txt is unchanged.*)

Pass by reference doesn't make a copy of the actual argument. Instead the formal argument is set to refer to the same memory location as the actual argument. (This is like one person having two phone numbers: no matter which number is called you get the same person.) **Pass by reference can be more efficient** than pass by copy, especially when passing large arrays or objects. However **pass by reference is dangerous**! It makes it possible for the code on one procedure to modify a variable in a

different procedure. This is sometimes called a ***side-effect*** and can make debugging much harder than otherwise.

With pass by reference, if the actual argument is not the name of some variable then the effect is the same as pass by copy. That is, there is no way to access the changed value once the procedure returns.

> (Technically you need to pass an expression called a **reference type** (or an *L-value*), since it is the sort of thing that can appear on the left hand side of an assignment statement. An expression that can appear on the right-hand side of an assignment statement is called a **value type** (or *R-value*).

(*Example of how pass by reference works: Make a short-cut to foo.txt, "mojo.txt". Make a change to mojo.txt and show how foo.txt was also changed. This is because you have only one file but with two names, or references.*)

With VB you indicate for each argument if it is passed by reference or by value (**`ByRef`** or **`ByVal`**) when you define the procedure. (You don't have a choice when you invoke a procedure, only when it is defined.)

Here's some examples:

```
Private Sub Square ( ByRef num As Integer )
   num = num * num
End Sub

Private Sub AddOne ( ByVal num As Integer )
   num = num + 1
End Sub

Private Sub AProcedure ()
   Dim x As Integer = 10
   AddOne ( x )
   MessageBox.Show( x )  ' shows 10
   Square ( x+1 )
   MessageBox.Show( x )  ' shows 10
   Square ( x )
   MessageBox.Show( x )  ' shows 100
End Sub
```

Note, **the names of the variables used for formal or actual arguments don't matter**, and can be the same name or different names. Since each name is local, they don't interfere with each other. (Somewhat more confusing is that when you pass a reference to some object, even by value, you can change the value of the properties of that object. This will be covered later in the course.)

> VB also allows optional arguments, but only at the end of the list of arguments. Each is proceeded with the `optional` keyword. You can also use `ParamArray args()` to pass a variable number of arguments.

Creating and Using Function Procedures

Whereas a Sub procedure does something, a Function procedure (or more simply, a "function") calculates something. The value of the function call is specified on a `Return` statement, which ends the function. (There is an `Exit Sub` command that can terminate a sub procedure call before the end is reached.) A function can be used as part of a calculation:

```
Private Function AddOne (ByVal num As Integer) _
     As Integer
   Return num + 1
End Function

Private Sub AProcedure ()
   Dim num As Integer = 10
   Dim next As Integer = AddOne ( num )
   MessageBox.Show( next )  ' shows 11
   num = AddOne( AddOne(num) )
   MessageBox.Show( num )  ' shows 12
```

Other than this difference, you can declare and pass parameters to functions just as will `Sub` procedures.

A common mistake is to use a Sub procedure with a `ByRef` argument named "result" or some such. Don't do that! **If your procedure calculates a value and needs to pass that value back, use a function.**

Avoid having procedures that both do something and calculate something. A procedure should do one task only!