

VARIABLES

Aim

Understanding how computer programs store values, and how they are accessed and used in computer programs.

WHAT ARE VARIABLES?

When you input data (i.e. information) into a computer the data is stored in the computer's memory. To visualise this, you might think of the computer's memory as millions of little boxes, each holding a single value. Normally, each box is numbered starting at zero. As a computer uses binary numbers (made up of 0s and 1s), the boxes may be numbered something like this: 10100010, which is 162 in decimal.

The actual number of boxes depends on the amount of memory in the specific computer. When data is put into the computer, the computer stores the data in the boxes that are part of the computer's Random Access Memory (RAM), and the number of those boxes is referred to as the memory address.

Obviously, for humans to refer to memory addresses by their binary index would be very difficult, so programming languages use a device called a variable. A variable is a word label that the programmer assigns to a piece of data, so they don't need to worry about where it is stored in memory or how to tell the computer where to find it.

Variables are simple memory boxes with names. You, the programmer, supply these names.

For example, the following lines first declare a variable called `myAge` of data type `Integer` (it will store whole numbers), and then assigns the integer value `25` to `myAge`:

```
Dim myAge As Integer  
myAge = 25
```

When you declare a variable, you must give it a name (and then compiler will claim some memory space and associate the name with the binary address), and specify the type of data that will be stored in that variable. The keyword `Dim` (short for dimension) instructs the compiler to claim some memory space, set its name, and define the data type that will be stored in it.

At this point, the variable will be empty, or more accurately, will contain a Null value. Null value exceptions can cause a lot of problems in programs and will be discussed in detail later. In the example above, the second line stores the value `25` in the variable named `myAge`. This is called initialising a variable. It is important to initialise variables as soon as you have declared them, to prevent the possibility of Null exceptions, even if you don't know what value it will eventually hold (e.g. you will be getting data from the user later in your program).

Variables can represent almost any value, and can be used in mathematical operations.

When naming your variables, there are a few rules you must follow to comply with the formal syntax for Visual Basic.NET. Variable names:

- Must start with a letter (a to z, A to Z)
- Can contain any number of letters or digits (a digit is 0 to 9)
- Can contain the underscore (e.g. _)
- Can be up to 255 characters long

Visual Basic .NET is not case sensitive. This means that once a variable named, say 'width' is declared, you cannot then declare another variable named, say 'Width', as to VB, they are considered identical. There are several different naming conventions used in programming; a popular style, Hungarian Notation, is described below. Make your variable names suggestive of what the variable represents, to increase readability of your code. Also, try to keep your names short and concise. It doesn't matter which naming style you decide to use, but be consistent, as changing styles throughout a program may make it hard to understand and maintain.

Because text display is so important in computing, Visual Basic.NET has a number of functions and commands that manipulate text. For example, you can join two or more strings into one, find the length of a string, or convert numbers to strings or strings to numbers (more about string operations later).

Arrays

An array is a collection of one or more variables of the same type, which allows you to store many values under a single variable name. An array's subscript identifies each element of the array. You can think of arrays as being similar to a series of post boxes for an apartment building. Each apartment has its own post box, which is numbered (this is the subscript of the array), and the whole collection of post boxes is given the street address of the apartment building (this is the variable name of the array).

To easily access each element of the array (say, within a loop) you can use a numerical variable for the array's subscript. An array subscript is placed between parentheses, i.e. (). By declaring an array with a `Dim` statement, VB will allocate the appropriate amount of memory for the array.

For example, the following line declares an array of Integers called `scores`, which can hold six separate `Integer` values:

```
Dim scores(5) As Integer
```

Notice that the size of the array (how many elements it can hold) is declared between the parentheses; this array can hold six elements, with indices numbered from 0 to 5. This is an important point to remember – indices are numbered from 0. Forgetting this will result in an "off by one" error.

Assigning values to the separate elements of an array is done by specifying the subscript, for example, this statement stores the value 30 in the third element of the `scores` array:

```
scores(2) = 30
```

You can also use a variable for the subscript. The following lines declare a variable to be used for the array index and assign it the value of 2. Then using the variable called `index`, assign the third element of the `scores` array the value of 5:

```
Dim index As Integer
index = 2
scores(index) = 5
```

HUNGARIAN NOTATION

Developed by Charles Simonyi from Microsoft, Hungarian Notation is a naming convention used to help programmers define and (more importantly) identify variable data types by a short abbreviation at the beginning of the variable name.

There are many advantages with using Hungarian Notation. These include:

- The ability to quickly and easily debug programs.
- The ability for teams of programmers to work on the same code without getting confused by the names of variables.
- Mnemonic value, allowing the programmer to easily remember the name of a variable.
- Efficient creation of variable names. By using Hungarian Notation, the time taken to ponder over an effective name for a variable is dramatically reduced.

Hungarian Notation uses a set of 'naming rules' to ensure that all of the above advantages (and many more) are realised:

- Convenient punctuation is used to define a variable type and name. For example `strName` represents a string variable that will contain data on a person's name.
- Simple data types are named by short tags that are chosen by the programmer. This ensures the variable names do not become too long, making them more prone to typographical errors which could result in bugs. A short tag also reduces the time it takes to type out the variable, making the programmer more efficient. For example, `intPhone` would be better to use than `integerPhone`. As the variable names are determined by the programmer, each programmer may differ on their preferred views of how to use Hungarian Notation. For example, one programmer may declare a variable as `curAccountbalance` and another may declare it as `currAccountBalance` (note the extra 'r' in curr). Both of these examples fit within the conventions of Hungarian Notation.

As you can see, Hungarian Notation is an extremely useful convention. It is not required when you program in VB (or any other language) but it is highly recommended to get into the habit of using the convention to make your programs easy to debug and modify (by yourself and other programmers). By understanding the different data types, you will realise how useful Hungarian Notation is.

By the way, the reason Hungarian Notation is named as such is due to the facts that Microsoft programmers who first saw it commented that the variables looked like they were in a language that was non-english and also that the creator Charles Simonyi was Hungarian.

KINDS OF VARIABLES (DATA TYPES)

There are different types of variables for different types of data. The different data types require different amounts of memory, and your compiler will allocate the appropriate amount when you declare variables. Below is a list of some of the most commonly used numerical variable data types:

Integer variables

The standard integer can store whole numbers in the range -32,768 to 32,767.

Long integer variables

A variable for storing large integers, whole numbers in the range -2,147,483,648 to 2,147,483,647.

Single precision floating point variables

Used to store fractions in the range of -3.402823E38 to 3.402823E38.

Double precision floating point variables

Another large variable used to store fractions with high accuracy; numbers in the range 1.79769313486232E308 and 1.79769313486232E308.

ASSIGNING VARIABLE VALUES

Just as you can assign a value to a variable, you can assign one variable's value to another. For example, look at the following piece of code:

```
intYourscore = 19000
intHighscore = intYourscore
```

The first statement sets `intYourscore` (an integer) to 19000.

The next statement, which is executed immediately after the first, assigns the value of `intYourscore` to `intHighscore`, i.e. it is a copy of the value in `intYourscore` – both variables would contain exactly the same value.

You can do more than that though. Say, you want to set one variable's value to be 100 more than another's. The following code does just that:

```
dblWorkerSalary = 1000
dblBossSalary = dblWorkerSalary + 100
```

From the above example `dblBossSalary` will equal 1100.

Now each time those two statements are executed, `dblBossSalary` (a double precision floating point variable) will be set to be 100 more than `dblWorkerSalary` (i.e. 1100). If `dblWorkerSalary = 300` in the first statement then `dblBossSalary` will equal 400 in the second statement.

Other mathematical operators could be used to produce the same or similar results, such as:

```
dblWorkerSalary = dblBossSalary - 100
```

This statement has the same effect as the previous example, only written differently.

```
dblBossSalary = dblWorkerSalary - 100
```

Sets `dblBossSalary` to be 100 less than `dblWorkerSalary` (an unlikely event).

```
dblBossSalary = dblWorkerSalary * 2
```

Sets `dblBossSalary` to be 2 times `dblWorkerSalary` (a more likely event).

```
dblBossSalary = dblWorkerSalary / 3
```

Sets `dblBossSalary` to be one third of `dblWorkerSalary` (never going to happen).

Now say you wish to increase a variable's value by 100. The following will increase `dblBossSalary` by 100:

```
dblBossSalary = dblBossSalary + 100
```

When the statement is executed, the value of `dblBossSalary + 100` is calculated first, and then the result is stored in `dblBossSalary`. This has the effect of increasing `dblBossSalary` by 100.

Any mathematical operator can be used, for example, the following will double the variable's value:

```
dblBossSalary = dblBossSalary * 2
```

For the most part, standard integers are used – other data types such as floating point variables are rarely used.

OPERATOR PRECEDENCE

Arithmetic operators are the basic set of operators used to perform mathematical calculations.

<i>Operator</i>	<i>Meaning</i>
<code>^</code>	exponent (power of)
<code>*</code>	multiply
<code>/</code>	division of doubles
<code>\</code>	division of integers
<code>Mod</code>	modulo (the remainder in a division, e.g. <code>3 Mod 2 = 1</code>)
<code>+</code>	addition
<code>-</code>	subtraction

The precedence of these operators means the order in which they are evaluated in a single statement. In the table above, exponent (`^`) has a higher precedence than multiplication (`*`) or division (`/`) which have the same precedence. Addition and subtraction have the lowest precedence, meaning they are evaluated last within a statement. Parentheses can be used to group calculations and ensure the correct evaluation of a statement. In the following two examples, the value of `Total` is different because of the parentheses used in example two.

- `Total = 2+5*3` results in `Total = 17`
- `Total = (2+5)*3` results in `Total = 21`

In example one above, the multiplication is evaluated first (i.e. multiplication has a higher precedence) and then the addition is calculated. The use of parentheses in the second example causes the addition to be evaluated first, even though normally addition is of lower precedence than multiplication.

STRINGS

Strings are variables that store text, words, phrases, sentences etc. If you type a letter in a word processor, the letter itself is held in a string variable in the computer's memory.

To store text in a string, you have to enclose it in quotes and assign it to a string variable. For example, the following statement will store the phrase `I'll be back` in the `strCliche` string variable:

```
strCliche = "I'll be back"
```

The string variable `strCliche` now contains the phrase `I'll be back`, without the surrounding quotation marks. The quotation marks are only there to indicate where the text string begins and ends. All literal strings need to be surrounded by quote marks.

As peculiar as it may sound, string variables can be added. Look at the following three statements:

```
strWord1 = "car"  
strWord2 = "pet"  
strWord3 = strWord1 + strWord2
```

The result is that `strWord3` would contain `carpet`.

Because string variables contain text and not numerical values, the two strings are joined together, one after the other, to form one string (i.e. in string math, `"car" + "pet" = "carpet"`). The process of adding strings together to form larger strings is called concatenation.

More than one string can be added at a time. For example, if the code were changed to:

```
strWord1 = "car"  
strWord2 = "pet"  
strWord3 = "My " + strWord2 + " got " + strWord1 + "sick."
```

`strWord3` would now contain `"My pet got carsick"`, because all those strings were joined together, one after the other. String variables (such as `strWord1`) and literal strings (such as `"got"`) may be added together.

Note the use of spaces – there is a space after `My` in the `"My "`, and one space before and one after `got` in the `" got "` string. These are put there to separate the text `"My"` from `"pet"` and `"got"` from `"pet"` and `"car"` respectively. There is no space in the `"sick"` string so that it forms up with the `car` to become `"carsick"`.

Note: Strings can ONLY be added and not subtracted. `strWord3 = strWord1 - strWord2` is meaningless and will only result in Visual Basic .NET giving an error message.

To set a string variable to nothing (i.e. remove all the text it contains), set it equal to "". For example, if you want to clear `strName`'s contents:

```
strName = ""
```

The quotes indicate the new text for `strName` is an empty string. The string variable `strName` will still exist, but it will contain nothing (a Null value).

HARD CODING VARIABLES

Usually when you write computer programs you do not know what the value of variables will be when the program will be run, i.e. the data will be entered by the user or read from a data source. However, there are also times when values used in your program are known right from the start, and not likely to change during the operation or lifetime of your program; a static value.

It might be tempting to simply use that value throughout your program code each time it is required in some calculation. But what happens if, at some point in the future, that value needs to be changed? Then you would have to read through your entire program, looking for that number and changing it. This would be difficult, time consuming and really boring for you as programmer – and the risk of making a mistake or missing an occurrence is quite likely.

In this case, it would be better to declare a variable and assign the static value to it. That way, if you ever need to change it in future, you would only need to go to the lines that declare and assign the variable, and change the value in that one place. This process is called hard coding variables.

A variable that is hard coded, and used in many places throughout your program, should be declared at the very start of the class or method in which it is used. By placing it at the top of your code you make it easy to find if you ever need to change it.

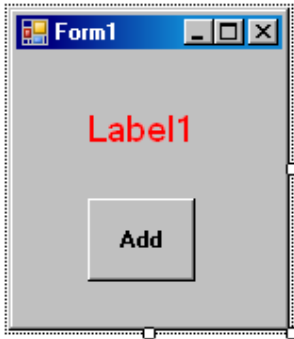
PROGRAMMING EXERCISE

Using Variables

You will now create a Windows based application that will simply add two numbers and display the result in a Window. The numbers will be hard coded, which means you will declare variables and assign values to them in the program code. The values you assign to the variables will not be changed; they will be set permanently within your program. In reality, this is not a very useful program, but writing this code will start you off in using variables, and introduce you to some of the other methods common in VB Windows applications.

1. Start a new VB Windows Application.

2. Add a label control and a button control to the form. Change the properties of the controls so they display any way you like – explore the options in the Properties panel and be creative; maybe add colour or change the font. Here's an example:



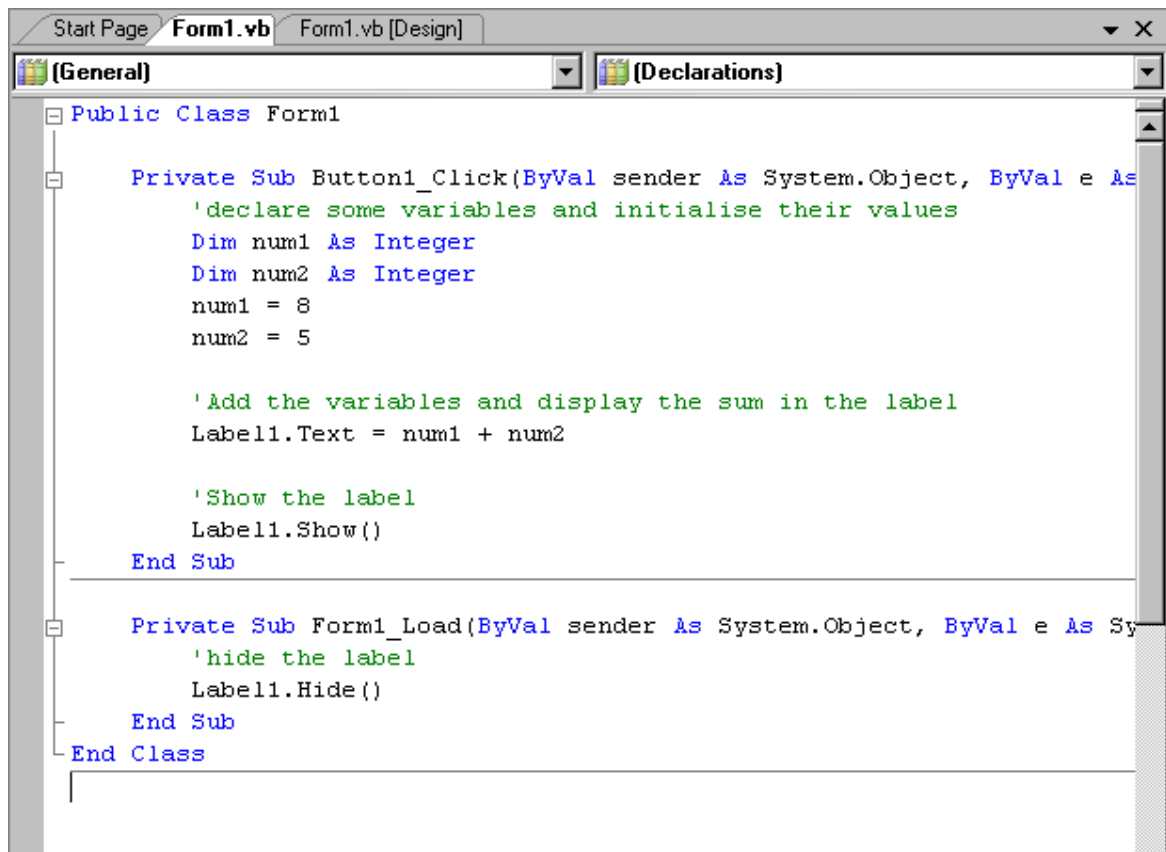
3. Double click on the form to display the `Form1_Load()` method. Add the following code:

```
'hide the label  
Label1.Hide()
```

4. Return to the form and double click on the Button to display the `Button1_Click()` method. Add the following code:

```
'declare some variables and initialise their values  
Dim num1 As Integer  
Dim num2 As Integer  
num1 = 8  
num2 = 5  
  
'Add the variables and display the sum in the label  
Label1.Text = num1 + num2  
  
'Show the label  
Label1.Show()
```


5. Remember to indent your code appropriately. Your program should look like this:



```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        'declare some variables and initialise their values
        Dim num1 As Integer
        Dim num2 As Integer
        num1 = 8
        num2 = 5

        'Add the variables and display the sum in the label
        Label1.Text = num1 + num2

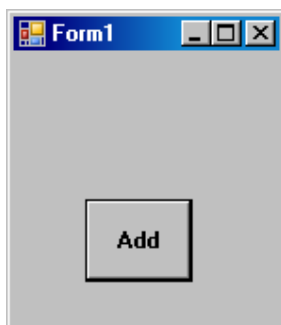
        'Show the label
        Label1.Show()
    End Sub

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        'hide the label
        Label1.Hide()
    End Sub
End Class
```

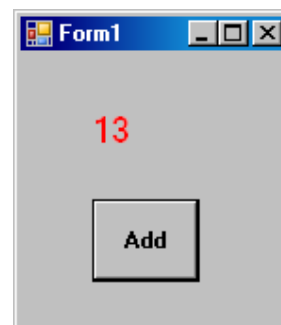
6. Save your files (both the form and the code page), then click the Run button on the toolbar:



7. When the application first runs, you should see:



- After clicking the Add button, you should see:



Comments

In this exercise you used comments to explain sections of the program code. A comment in VB is a line started with an apostrophe (i.e. ') and displayed in green text in the IDE. The compiler will ignore comments when building the runtime code (more about runtime code in a later lesson).


Comments are important as they help programmers document their programs, and provide explanation of what the code is designed to do. As your programs become more complicated, comments will be increasingly important. Also, if you ever need to revisit a piece of code you wrote a long time ago, they will help you remember what the code involves. Likewise, comments provide important information to other programmers who a may need to amend or extend your programs.

It is best to get into the habit of writing comments as you write program code. Although many of the programs you will write during this course do not really require commenting, you should begin now to always include them in your programs. Unfortunately, good commenting is an often missing component in program writing today. By being diligent in including meaningful comments, you can set yourself apart from many of the programmers currently working, and display a professionalism in your craft.

8. Go back to the `Form1_Load()` method, and type an apostrophe in front of the `Label1.Show()` code. This makes this line a comment, but keeps it in the source program code, in case you wish to put it back in later. Save your project and then run it again. Notice the difference?


SET READING

Spend approximately an hour reading about using Visual Basic .NET Variables and Arrays, and how to perform calculations, on some of the websites listed in Appendix 2 Visual Basic .NET Resources, on other websites you find helpful, or in any reference text books you have to hand.

	SELF ASSESSMENT Perform the self assessment test titled Self Assessment Test 2. If you answer incorrectly, review the notes and try the test again.
---	--

SET TASK

Extend the program you wrote in this lesson by adding more text labels, which will display the result of subtracting, multiplying and dividing the variables. Add all the necessary code to the `Button1_Click()` method to perform those arithmetic operations and display the results in the new text labels. Be sure to use the correct data types for your variables. Experiment a little; create your own variation for this program, and don't forget to include comments.

	ASSIGNMENT Download and do the assignment called Assignment 2.
---	--