Ah.Fawad " Saiq "

Visual Basic .NET Programming for Beginners

This Home and Learn computer course is an introduction to Visual Basic.NET programming for beginners. This course assumes that you have no programming experience whatsoever. It's a lot easier than you think, and can be a very rewarding hobby!

You don't need to buy any software for this course! You can use the new FREE Visual Basic Express Edition from Microsoft. You can download it here:

Getting Started with Visual Basic .NET

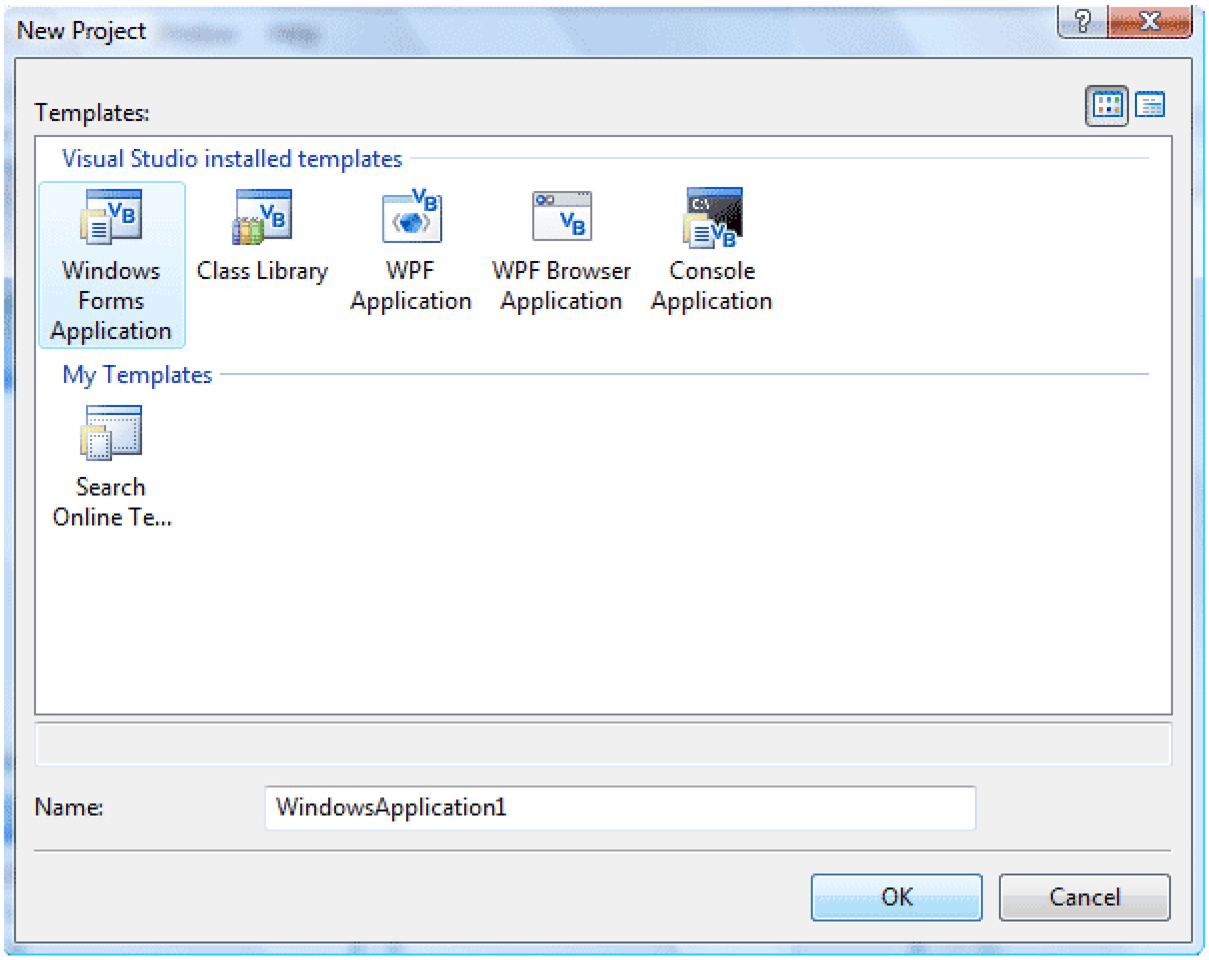
Make a Start with VB .NET

Launch your Visual Basic .NET or Visual Studio software. When the software first loads, you'll see a screen something like this one:

Click here to see the image in a new popup window (86K)

There's a lot happening on the start page. But basically, this is where you can start a new project, or open an existing one. The first Tab, Projects, is selected. At the moment, the area labelled "Open an Existing Project" is blank. This is what you'll see when you run the software for the first time (because you haven't created a project yet). When you create a project, the Name you gave it will be displayed on this page, as a hyperlink. Clicking the link will open the project.

At the bottom of the screen, there are two buttons: "New Project" and "Open Project". To get started, click the "New Project" button. When you do, you'll see this dialogue box appear:



Name: WindowsApplication1

As a beginner, you'll normally want the option selected: "Windows Application", in the "Visual Basic Projects" folder. This means that you're going to be designing a programme to run on a computer running the Microsoft Windows operating system.

If you look in the Name textbox at the bottom, you'll see it says "WindowsApplication1". This is the default name for your projects. It's not a good idea to keep this name. After all, you don't want all of your projects to be called "WindowsApplication1", "WindowsApplication2", etc. So click inside this textbox and change this Name to the following:

My First Project

Keep the Location the same as the default. This is a folder inside of your "My Documents" folder called "Visual Studio Projects". A new folder will then be created for you, and its name will be the one you typed in the "Name" textbox. All of your files for your first project are then saved in this folder.

Click the OK button, and the Visual Basic NET design time environment will open. It will look like the following (the 2008 edition is just the same):

Click here to see the image in a new popup window (28K)

That's a very daunting piece of software, hey? Well, don't worry. We'll break it down bit by bit in the next few sections, and pretty soon you'll be zipping your way around it like a pro!

Visual Basic .NET Forms

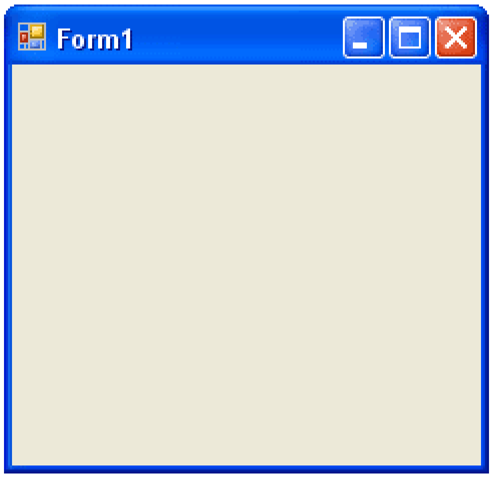
The Default Form

In the Visual Basic NET design time environment, the first thing to concentrate on is that strange, big square in the top left. That's called a form. It's actually the pretty bit of your programme, the part that others will see when they launch your masterpiece. Granted, it doesn't look too attractive at the moment, but you'll soon discover ways to lick it into shape.

To run the form, try this:

- From the menu bar, click **Debug**
- From the drop down menu, click **Start**
- Alternatively, press the **F5** key on your keyboard
- Your programme is launched

Congratulations! You have now created your very first programme. It should look like this:



Click the Red X on the form to stop it from running. You will then be returned to the software environment.

If you compare the first form with the one above, you'll see that they look very similar. But the one above is actually a real programme, something you could package and sell to unsuspecting village idiots

So what's going on? Why the two different views? Well, Visual Basic has two distinct environments, a **Design** environment and a **Debug** environment. Design Time is where you get to play about with the form, spruce it up, add textboxes, and buttons, and labels (and code, of course); Debug is where you can test your programme and see how well it performs. Or doesn't perform, as is usually the case.

But don't worry about the terminology, for the time being. Just be aware that there's a two step process to VB programming: **designing** and **debugging**.

So, let's get on and do some designing! Before we can start designing a form, though, we need some tools. And where are tools kept? In a toolbox!

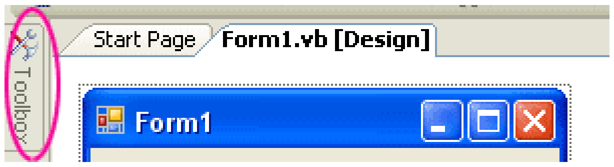
Click below for the next lesson (unless you fancy buying our VB .NET book!).

dding Controls Using the Toolbox

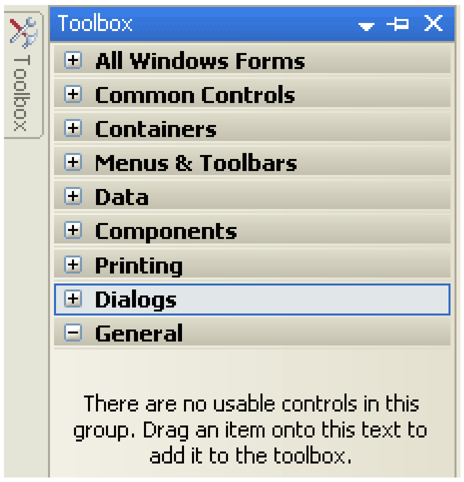
Adding a Control to a Form

Things like buttons, textboxes, and labels are all things that you can add to your Forms. They are know as Controls, and are kept in the Toolbox for ease of use.

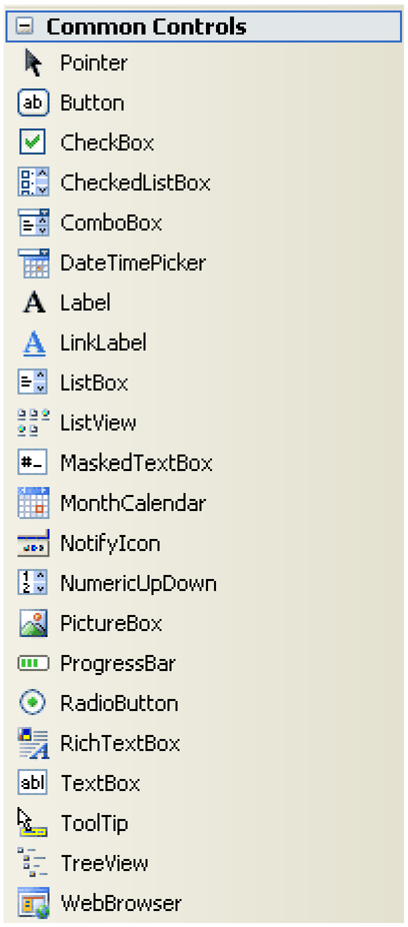
The Toolbox can be found on the left of the screen. In the picture below, you can see the toolbox icon next to Form1:



To display all the tools, move your mouse over the toolbox icon. You'll see the following automatically appear:



There are seven categories of tools available. The toolbox you'll be working with first is the Common Controls toolbox. To see the tools, click on the plus symbol next to **Common Controls**. You'll see a long list of tools:



As you can see, there are an awful lot of tools to choose from! For this first section, we'll only be using the **Button**, the **TextBox** and the **Label**.

If you want to keep the toolbox displayed, click the Pin icon next to the X. To close the toolbox, simply move your mouse away.

In the next part, we'll see how to add a textbox to the form.

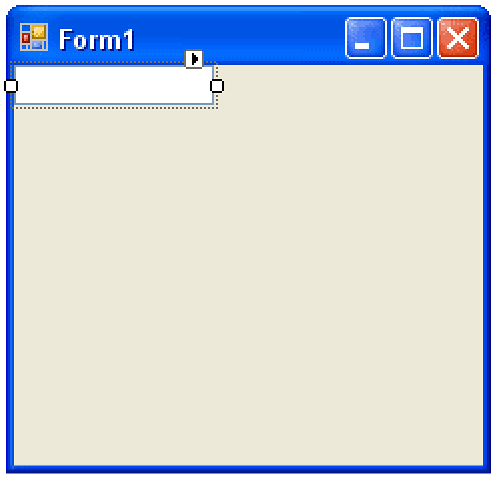
Adding a Tool (Control) to your Form

How to Add a Control to your VB .NET Forms

Let's start by adding a textbox to our form. With the tools displayed, do the following:

- Locate the TextBox tool
- Double click the icon
- A textbox is added to your form

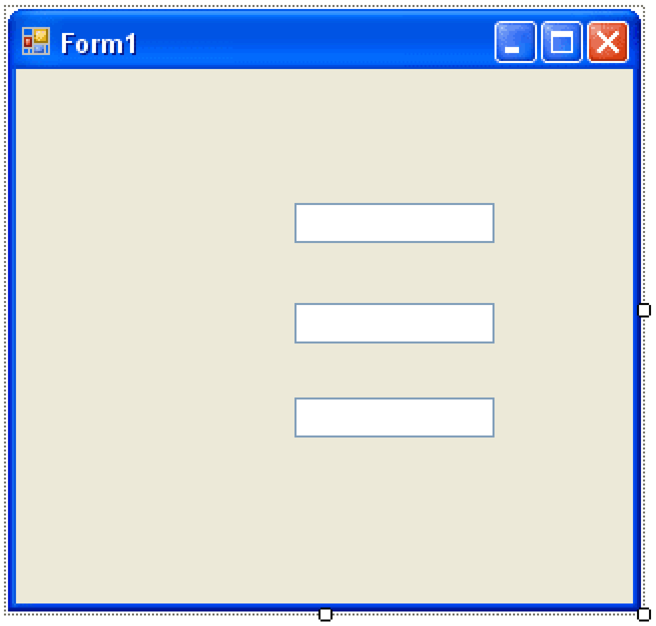
The textbox gets added to the top left position of your form. To move it down, hold your mouse over the textbox and drag to a new position:



Notice the small squares around the textbox. These are sizing handles. Move your mouse over one of them. The mouse pointer turns into an extended line with arrowheads. Hold your left mouse button down and drag outwards. The textbox is resized. Play around with the sizing handles until you're happy with the size of your textbox.

One thing you will notice is that you can't make the size any higher, but you can make it wider. The reason why you can't make it any higher is because the default action of a textbox is to have it contain only a single line of text. If it's only going to contain one line of text, Microsoft reasoned, there's no reason why you should be able to change its height. A textbox can only be made higher if it's set to contain multiple lines of text. You'll see how to do this soon.

- Create two more textboxes by double clicking on the textbox icon in the toolbar (Or Right-click on the selected textbox and choose Copy. Then Right-click on the Form and choose Paste.)
- Resize them to the same size as your first one
- Line them up one below the other with space in between
- Try to create something that looks like the one below

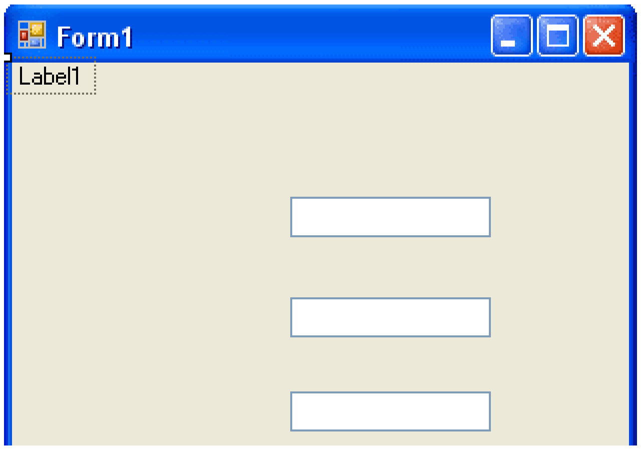


No more reading these lessons online - get the eBook here!

Adding a Label to your Form

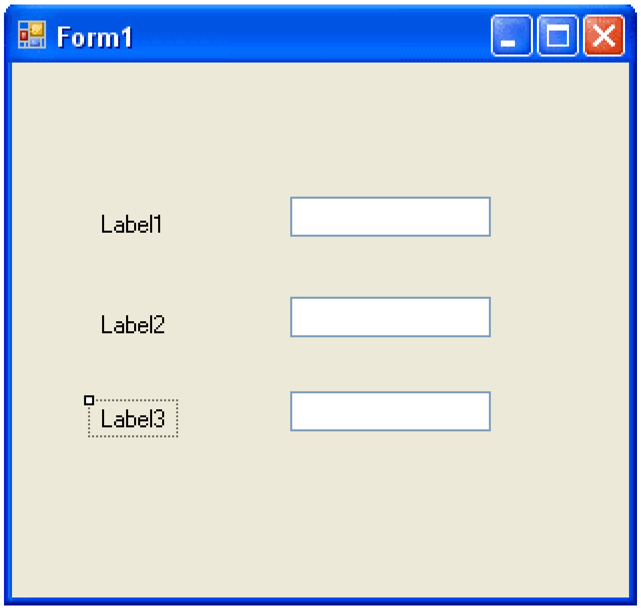
Let's add some labels near the textboxes so that your users will know what they are for.

- Locate the label control in the toolbox
- Double click the label icon
- A new label is added to your form
- It should look like the one below

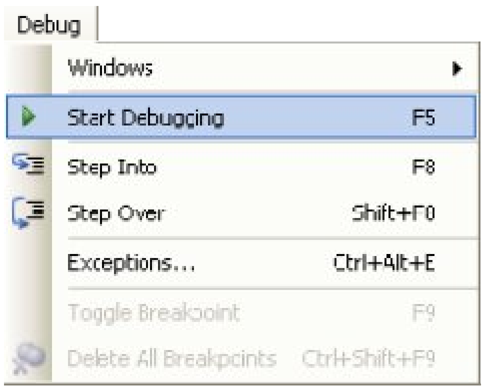


Click on the label to select it. Now hold your left mouse button down on the label. Keep it held down and drag it to the left of the textbox.

Create two more labels, and position them to the left of the textboxes. You should now have a form like this one:

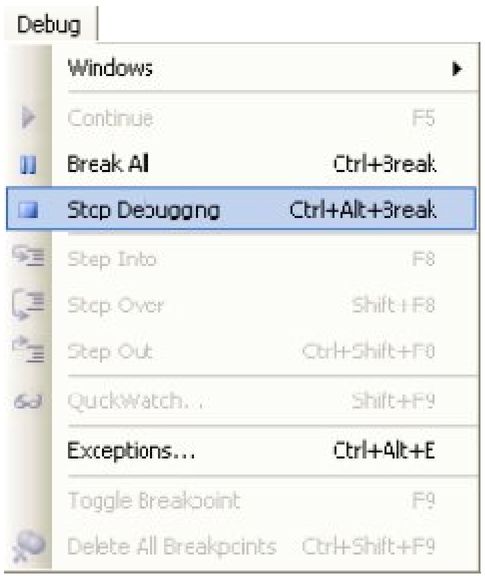


To see what your Form looks like as a programme, click **Debug > Start** from the menu bar. Or press F5 on your keyboard:

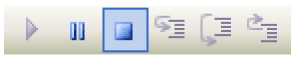


To stop the programme from running, you can do one of the following:

1. Click the Red X at the top right of your Form
2. Click **Debug > Stop Debugging** from the menu bar
3. Press Shift + F5 on your keyboard



You can also click the Stop button on the VB toolbars at the top, as in the image below:



All right, we're getting somewhere. We now have a form with textboxes and labels, something that looks like a form people can fill in. But those labels are not exactly descriptive, and our textboxes have the default text in them. So how can we enter our own text for the labels, and get rid of that default text for the textboxes?

To do those things, we need to discuss something called a Property.

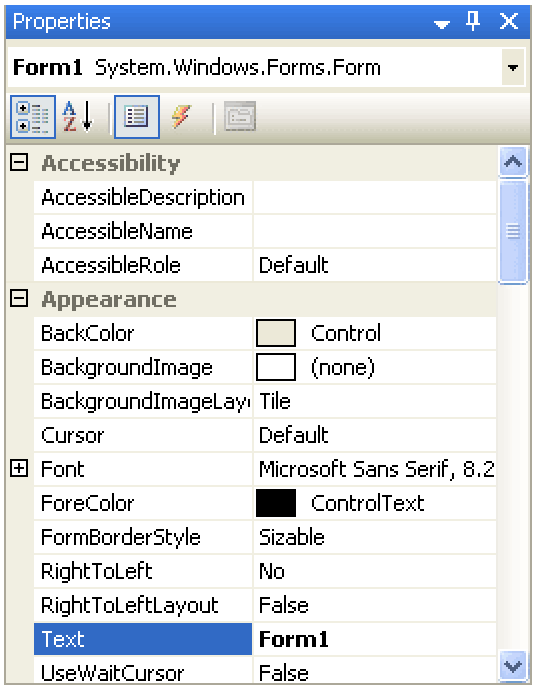
Properties

An Introduction to VB .NET Properties

You've probably noticed the area to the right of the design environment, the area with all the textboxes in a grid, the one that has those daunting names like "AccessibleDescription", "AccessibleName", "AccessibleRole". That's the Properties box.

Click anywhere on the form that is not a label or a textbox, somewhere on the form's grey areas. The form should have the little sizing handles now, indicating that the form is selected.

On the right of the design environment there should be the following Properties box:



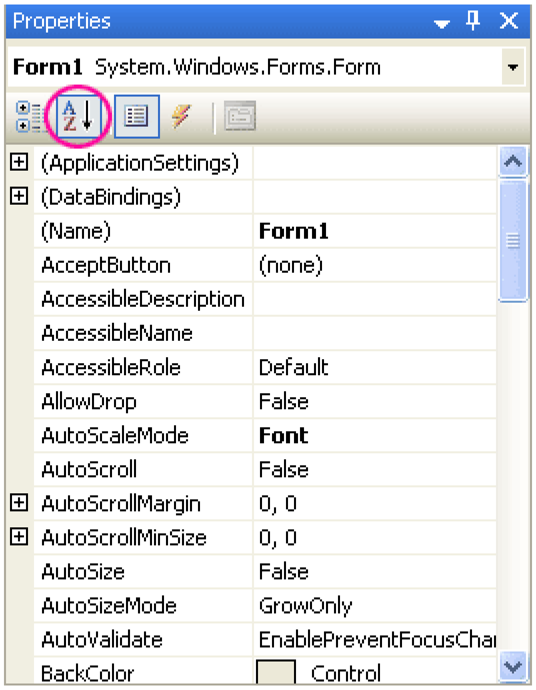
Form1 System.Windows.Forms.Form

⊟ Accessibility	
AccessibleDescription	
AccessibleName	
AccessibleRole	Default
⊟ Appearance	
BackColor	▭ Control
BackgroundImage	▭ (none)
BackgroundImageLay	Tile
Cursor	Default
⊞ Font	Microsoft Sans Serif, 8.2
ForeColor	■ ControlText
FormBorderStyle	Sizable
RightToLeft	No
RightToLeftLayout	False
Text	**Form1**
UseWaitCursor	False

If your Properties box says "Textbox1 Textbox" or "Label1 Label" then you haven't yet selected the Form. Click away from the textbox or label until the Properties box reads "Form1 Form"

What you are looking at is a list of the properties that a form has: Name , BackColor, Font, Image, Text, etc. Just to the right of these properties are the values for them. These values are the default values, and can be changed. We're going to change the value of the Text property.

First, you might want to display the list of Properties in a more accessible form. You can display the list properties alphabetically. To do that, click the Alphabetic icon at the top of the Properties box, as in the image below:



⊞ (ApplicationSettings)	
⊞ (DataBindings)	
(Name)	**Form1**
AcceptButton	(none)
AccessibleDescription	
AccessibleName	
AccessibleRole	Default
AllowDrop	False
AutoScaleMode	**Font**
AutoScroll	False
⊞ AutoScrollMargin	0, 0
⊞ AutoScrollMinSize	0, 0
AutoSize	False
AutoSizeMode	GrowOnly
AutoValidate	EnablePreventFocusChai
BackColor	Control

This will make the properties easier to find.

Before we change any in the Properties box, let's clear up what we mean by "Property".

No more reading these lessons online - get the eBook here!

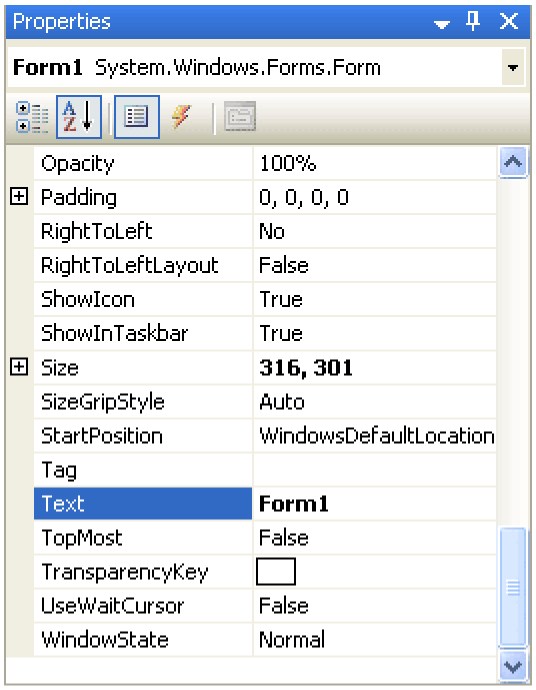
What is a Property?

Those controls you added to the form (textboxes and labels), and the form itself, are called control objects. You can think of controls as things, something solid that you can pick up and move about. Controls (things) have properties. If your television were a control, it too would have properties: an On/Off button property, a colour property, a volume property, and a ... well, what other properties would your television have? Think about it.

The properties of your television will have values. The On/Off button would have just two values - On or Off. The volume property could have a range of values, from zero to ten, for example. If the value of the volume property was set to ten, the loudest value, then you'd probably have some very angry neighbours!

In VB.NET, you can change a property of a control from the Properties Box. (You can also change a property using code, which you'll do quite a lot.) If we go back to our Form object, and the properties and values it has, we can see how to change them using the Properties Box. We'll change only one of these values for now - the value of the Text property . So, do this:

- Locate the word "Text" in the Property box, as in the image below



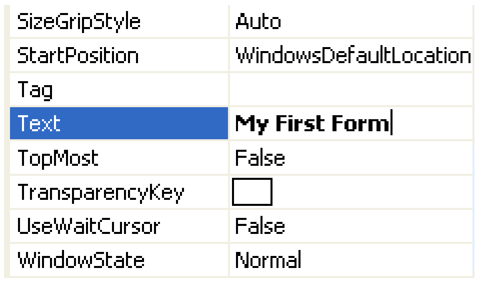
Properties

Form1 System.Windows.Forms.Form

Opacity	100%
⊞ Padding	0, 0, 0, 0
RightToLeft	No
RightToLeftLayout	False
ShowIcon	True
ShowInTaskbar	True
⊞ Size	**316, 301**
SizeGripStyle	Auto
StartPosition	WindowsDefaultLocation
Tag	
Text	**Form1**
TopMost	False
TransparencyKey	⬜
UseWaitCursor	False
WindowState	Normal

"Text" is a Property of Form1. Don't be confused by the word "Form1" next to the word "Text". All this means is that the current value of the Text property is set to the word "Form1". This is the default.
To change this to something of your own, do this:

- Click inside the area next to "Text", and delete the word "Form1" by hitting the backspace key on your keyboard
- When "Form1" has been deleted, type the words "My First Form"

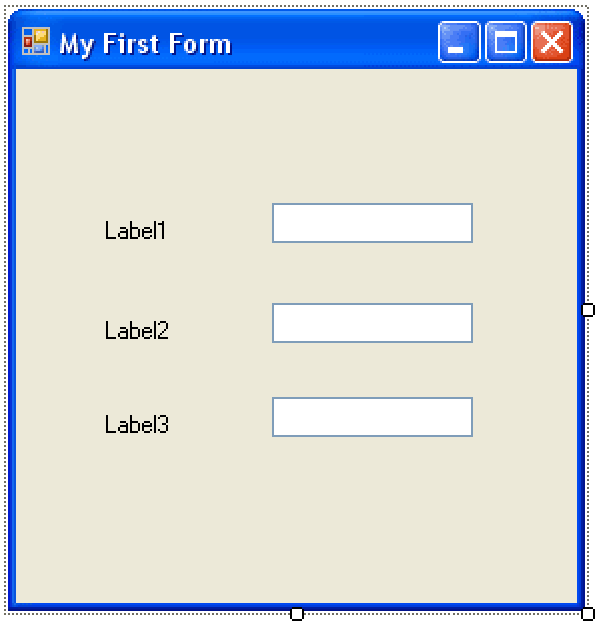


SizeGripStyle	Auto
StartPosition	WindowsDefaultLocation
Tag	
Text	**My First Form**
TopMost	False
TransparencyKey	⬜
UseWaitCursor	False
WindowState	Normal

- Click back on the form itself (the one with the labels and textboxes), or hit the return key on your keyboard

- The words "My First Form" will appear as white text on a blue background at the top of the form

When you've correctly changed the Text property, your Form will then look like this one:



As you can see, your new text has gone at the top of the form, in white on the blue background.

So the Text Property of a form is for setting the caption you want to display in the title bar at the top.

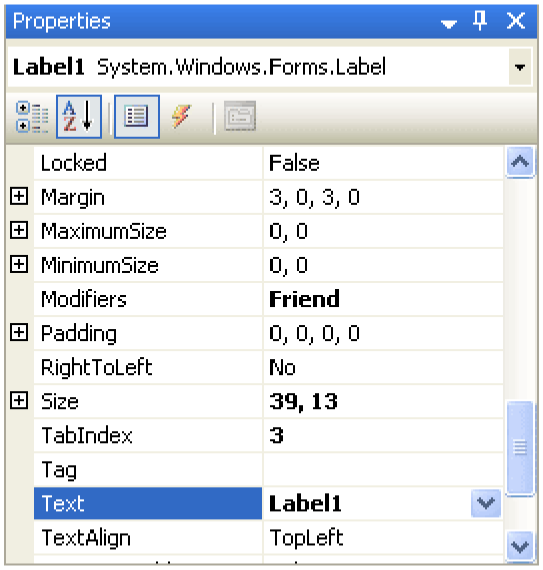
In the next part, we'll take a look at how to change the text property of labels and textboxes.

The Text Property

The Text Property of a Control

Changing the values of some properties is fairly easy. We'll now change the Text properties of our labels, and the Text properties of our Textboxes.

Click on Label1 so that it has the sizing handles, and is therefore selected. Examine the Property box for the Label:

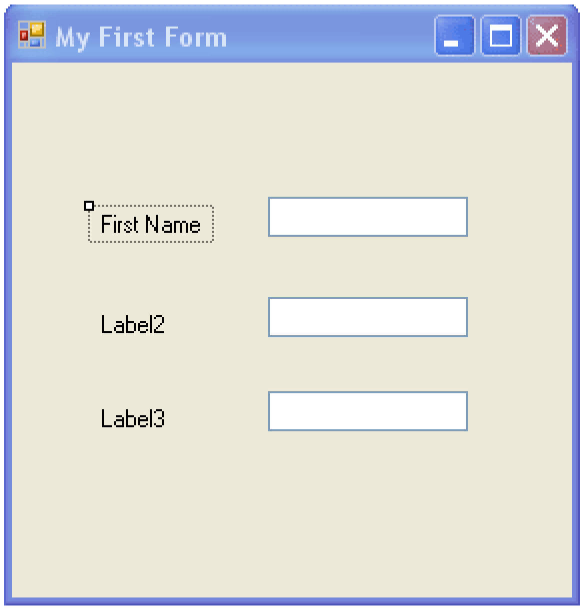


Locked	False
⊞ Margin	3, 0, 3, 0
⊞ MaximumSize	0, 0
⊞ MinimumSize	0, 0
Modifiers	**Friend**
⊞ Padding	0, 0, 0, 0
RightToLeft	No
⊞ Size	**39, 13**
TabIndex	**3**
Tag	
Text	**Label1**
TextAlign	TopLeft

You can see that the Label control has quite a few different properties to the Form control. Think back to your television as an control. It will have different buttons and knobs to your DVD Player control. A label has different "buttons and knobs" to the Form's "buttons and knobs".

But the Label has a lot of properties that are the same. The Text property of a Label does what you'd expect it to do: adds text to your label. We'll do that now:

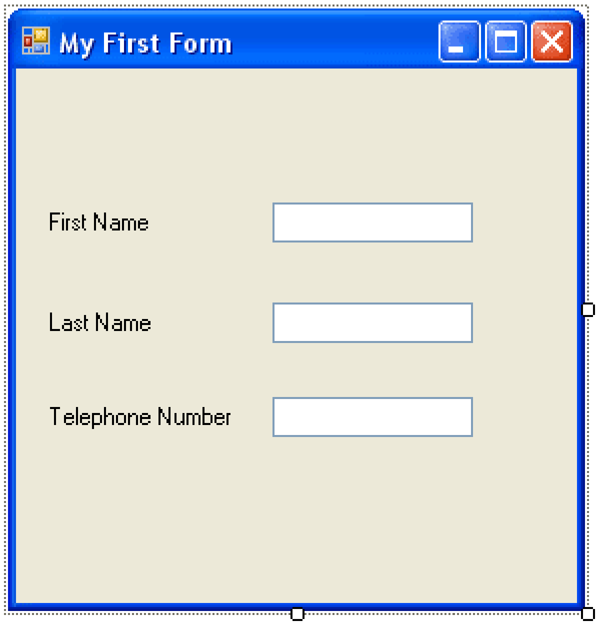
- With label1 selected, click inside the area next to "Text", and delete the word "Label1" by hitting the backspace key on your keyboard
- Type in the words "First Name"
- Click back onto the grey form, or hit the return key on your keyboard
- Label1 has now changed its text caption to read "First Name"
- If you've made a typing error, go back to the first step above and try again
- Your form should now look like this:



Now, change the Text property of the other two labels. Change them to these values:

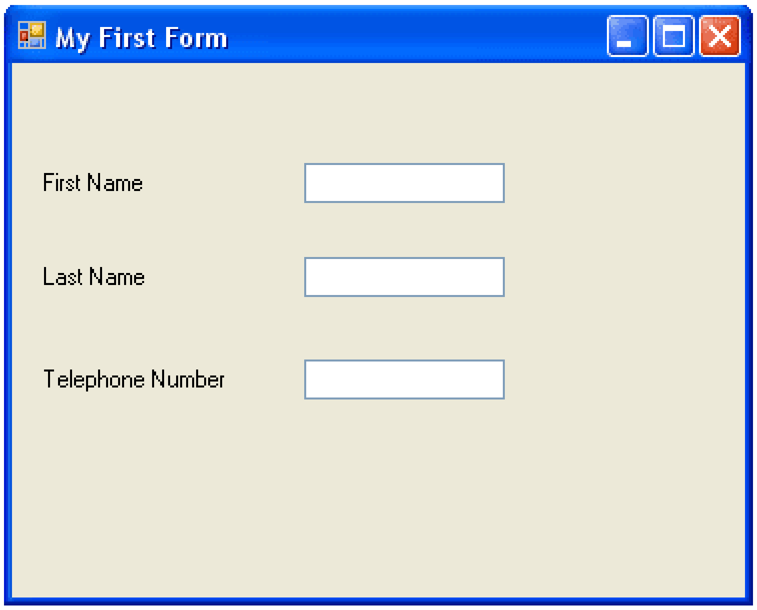
Label2: Last Name
Label3: Telephone Number

What you should notice is that the labels resize themselves, after you press the enter key to commit the changes. You may need to drag your labels to the left a bit. But when you're finished, your form should look like ours below:



The form might look a little squashed, though. Is there anything we can do to make it bigger? Well, it just so happens there is.

The Form can be resized just like the Label and the textboxes. Click anywhere on the form that is not a textbox or a label. If you look closely around the Form's edges, you'll notice our old friends the sizing handles. To make the form bigger, just stretch them like you did the labels and the textboxes. Play around with the size of the form until you're happy with it. You can now reposition and resize the textboxes and labels so that things don't look too squashed. Your form might look like this one:



Click on **Debug > Start** to have a look at your programme. Or Press F5 on your keyboard. Click **Debug > Stop Debugging** to get back to the design environment. (Or press Shift + F5, or just click the red X at the top right of the form.)

Now, lets add a splash of colour to our form.

Adding a Splash of Colour

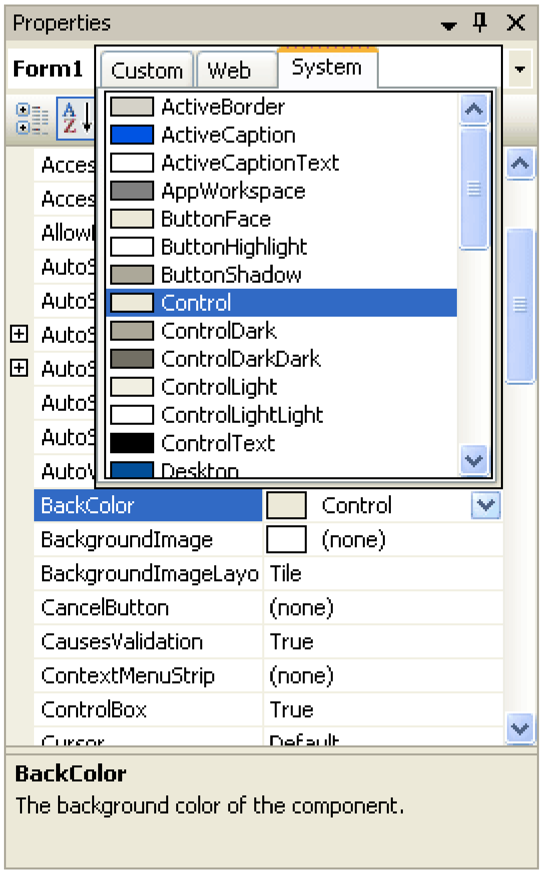
Liven up your VB .NET Forms

At the moment, our form looks a little bland. Time to liven it up with a splash of colour.

Changing the colour of the Form means we have to change one of its properties - the BackColor property.

So click anywhere on the form that is not a textbox or a label. If you do it right, you should see the sizing handles around the edges of the grey form. The Property Box on the right will read "Form1", and that indicates that you have indeed selected the form. When the Form is selected you can change its properties.

To change the colour of the Form, click the word "BackColor" in the Property Box. Next, click the black down-pointing arrow to the right. A drop-down box will appear.



ActiveBorder
ActiveCaption
ActiveCaptionText
AppWorkspace
ButtonFace
ButtonHighlight
ButtonShadow
Control
ControlDark
ControlDarkDark
ControlLight
ControlLightLight
ControlText
Desktop

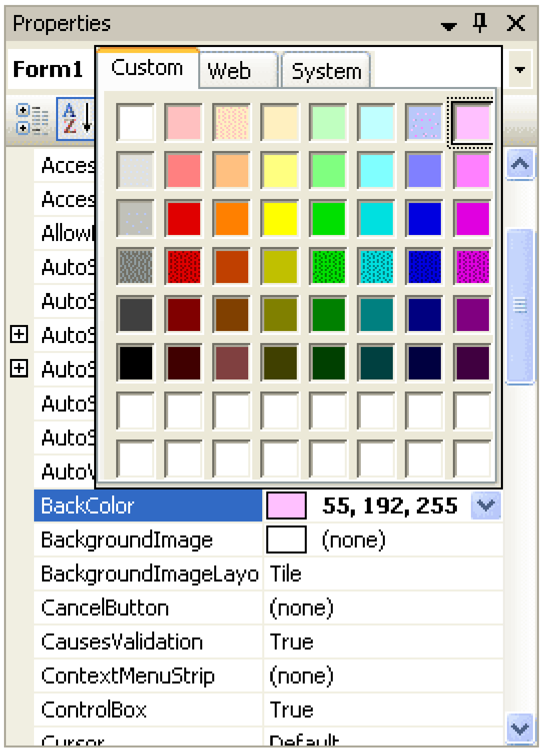
BackColor	Control
BackgroundImage	(none)
BackgroundImageLayo	Tile
CancelButton	(none)
CausesValidation	True
ContextMenuStrip	(none)
ControlBox	True
Cursor	Default

BackColor
The background color of the component.

The default colour is the one selected - **Control**. This is on the **System** Tab. The System colours are to set whatever colour scheme the user has opted for when setting up their computers. For example, you can use the Display Properties dialogue box in Windows XP to change how things like menus and buttons look. Someone who is colour-blind might have changed his or her settings in order to see things better on the computer screen. If you stick with the System colours then a colour-blind user of your programme would not have any problems seeing your master work.

As you can see in the image above, you can choose the colour of the Active Caption. The Active Caption is the one you set earlier when you changed the text to "My First Form". The Active Caption is blue on my computer, and the Active Caption Text is white. It might be different on yours.

If you want to choose a colour that is not a System colour, click the Custom Tab. You'll then see this:



BackColor	55, 192, 255
BackgroundImage	(none)
BackgroundImageLayo	Tile
CancelButton	(none)
CausesValidation	True
ContextMenuStrip	(none)
ControlBox	True

Click on any of the Colours in the colour palette and the background colour of your form will change.

You can also select the Web Tab. When you do, you'll see a list of Web-Safe colours to choose from. A Web-Safe colour is one that displays correctly in a web browser, regardless of which computer being used (that's the theory, anyway). You might want to use a Web-Safe colour if you're designing a project for the internet. But you can choose one even if you're not.

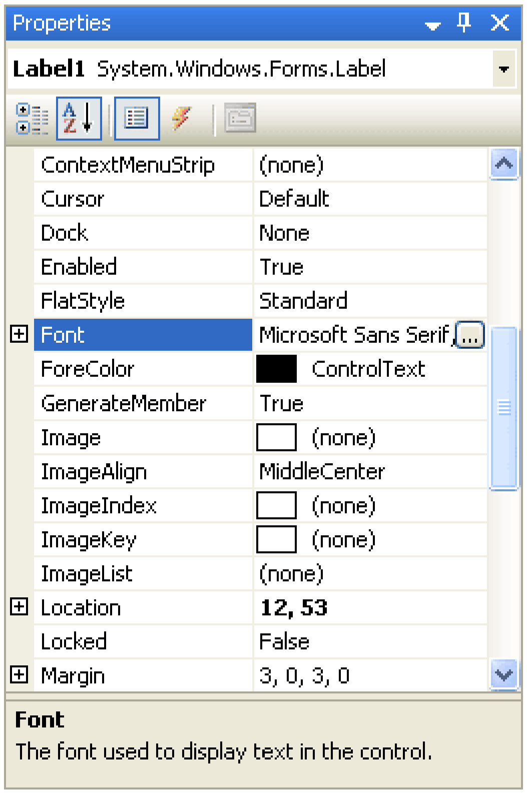
To change the colour of the labels, click on a label to select it. Look in the Property box to see if it reads Label. If so, you can now go ahead and change the BackColor property of the Label in exactly the same way that we changed the BackColor property for our Form.

Change the colour of the other two labels to anything you like. To change the colour of more than one Label at a time, click on one Label to select it. Now, hold down the "Ctrl" key on your keyboard and click another Label. You'll see that two Labels now have sizing handles around them. Click the third Label with the "Ctrl" key held down, and all three Labels will be selected. You can change the BackColor property of all three at once.

If you want to change the Font size of the Labels and Textboxes, select a control. Let's start with Label1.

- So click on Label 1
- Scroll down the Property Box until you see Font
- Click on the word "Font" to highlight it
- MS Sans Serif is the default Font

Notice that the Font property has a cross next to it. This indicates that the property is expandable. Click the cross to see the following:

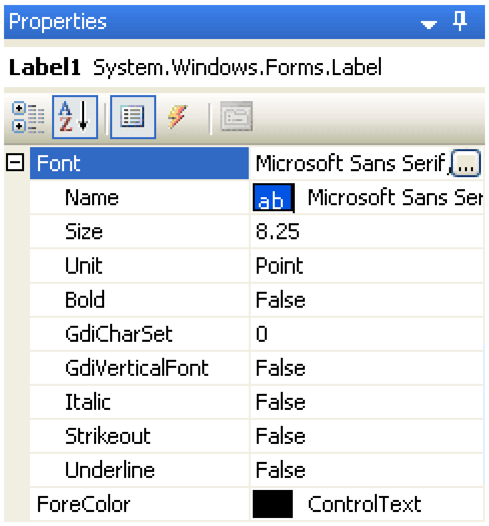


Label1 System.Windows.Forms.Label

ContextMenuStrip	(none)
Cursor	Default
Dock	None
Enabled	True
FlatStyle	Standard
⊞ Font	Microsoft Sans Serif,
ForeColor	■ ControlText
GenerateMember	True
Image	□ (none)
ImageAlign	MiddleCenter
ImageIndex	□ (none)
ImageKey	□ (none)
ImageList	(none)
⊞ Location	**12, 53**
Locked	False
⊞ Margin	3, 0, 3, 0

Font
The font used to display text in the control.

Notice that the Font property has a cross next to it. This indicates that the property is expandable. Click the cross to see the following:


Properties

Label1 System.Windows.Forms.Label

Font	Microsoft Sans Serif
Name	Microsoft Sans Ser
Size	8.25
Unit	Point
Bold	False
GdiCharSet	0
GdiVerticalFont	False
Italic	False
Strikeout	False
Underline	False
ForeColor	■ ControlText

As you can see, you can change a lot of Font properties from here: the Name of the font, its Size, whether is should be Bold or not, etc. You can also click the square box with the three dots in it. This brings up a dialogue box where you can change the font properties in the same place.

Make the following changes to the three labels:

Font: Arial
Font Style: Bold
Font Size: 10

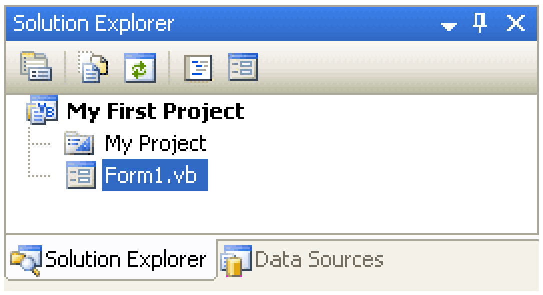
Change the Font of the three Textboxes so that they are the same as the Labels.

When you're finished, you should have a form that looks a little more like a real programme. Time now to save your work. Click below to see how to do this in VB .NET.

Saving your work

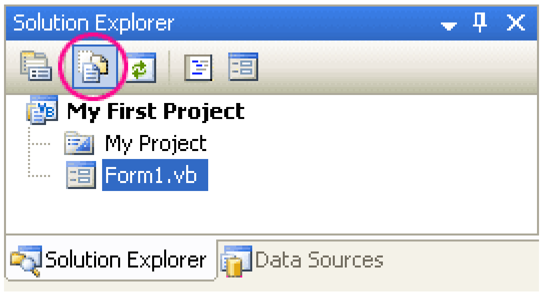
How to Save your VB .NET Projects

If you have a look in the top right of the Design Environment, you'll see the Solution Explorer. (If you can't see it, click **View > Solution Explorer**.)

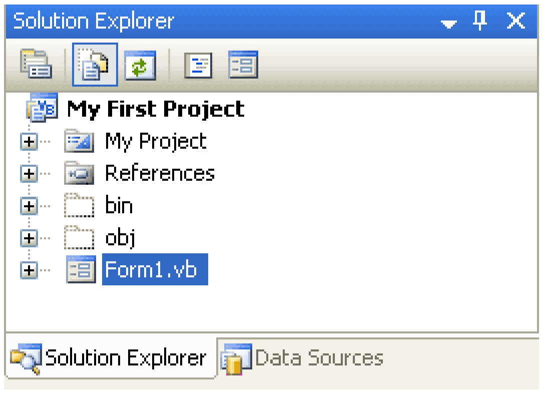


The Solution Explorer shows you all the files you have in your project (Notice that the name of your project is at the top of the tree - "My First Project").

At first glance, it looks as though there are not many files in the project. But click the Show All Files icon, circled below:

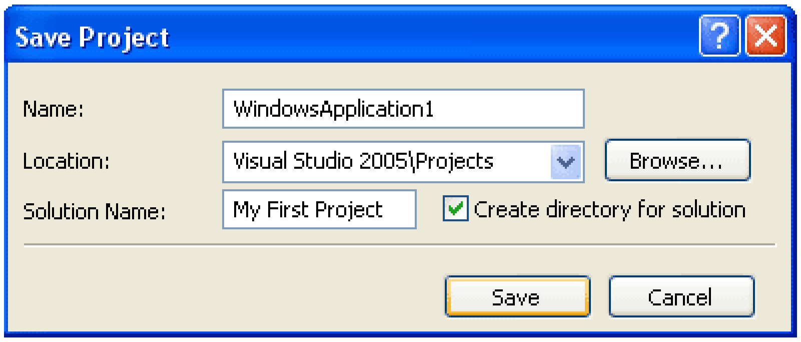


When you click Show All Files, the Solution Explorer will look something like this:



When you save your project, you are saving all these files.

To save your work, click **File > Save All** and you'll see the following dialogue box (we've chopped ours down a bit):



Name: WindowsApplication1
Location: Visual Studio 2005\Projects
Solution Name: My First Project
[✓] Create directory for solution

The files are usually saved in the My Document folder in XP (Document folder in Vista), under Visual Studio. If you want to save your projects elsewhere, click the Browse button.

To actually save your work as you go along, just click File > Save All from the menu bar. Or press Ctrl + Shift + S on your keyboard. Or click the icon in the Toolbar (the stack of floppy disks). If you save often then you won't lose any of your work if anything goes wrong with your computer.

In the next section, we'll get down and do a bit of actual programming. It's a gentle introduction, so nothing to get too worried about!

How to Create Variables in VB .NET

VB .NET Variables

Why are we discussing variables? And what is a variable?

With Visual Basic, and most programming languages, what you are doing is storing things in the computer's memory, and manipulating this store. If you want to add two numbers together, you put the numbers into storage areas and "tell" Visual Basic to add them up. But you can't do this without variables.

So a variable is a storage area of the computer's memory. Think of it like this: a variable is an empty cardboard box. Now, imagine you have a very large room, and in this room you have a whole lot of empty cardboard boxes. Each empty cardboard box is a single variable. To add two numbers together, write the first number on a piece of paper and put the piece of paper into an empty box. Write the second number on a piece of paper and put this second piece of paper in a different cardboard box.

Now, out of all your thousands of empty cardboard boxes two of them contain pieces of paper with numbers on them. To help you remember which of the thousands of boxes hold your numbers, put a sticky label on each of the two boxes. Write "number1" on the first sticky label, and "number2" on the second label.

What have we just done? Well, we've created a large memory area (the room and the cardboard boxes), and we've set up two of the boxes to hold our numbers (two variables). We've also given each of these variables a name (the sticky labels) so that we can remember where they are.

Now examine this:

```
Dim number1 As Integer
Dim number 2 As Integer

number1 = 3
number2 = 5
```

That's code from Visual Basic Net. It's VB's way of setting up (or declaring) variables.

Here's a breakdown of the variable Declaration:

Dim

Short for Dimension. It's a type of variable. You declare (or "tell" Visual Basic) that you are setting up a variable with this word. We'll meet other types of variables later, but for now just remember to start your variable declarations with Dim.

number1

This is the cardboard box and the sticky label all in one. This is a variable. In other words, our storage area. After the Dim word, Visual Basic is looking for the name of your variable. You can call your variable almost anything you like, but there are a few reserved words that VB won't allow. It's good practice to give your variables a name appropriate to what is going in the variable.

As Integer

We're telling Visual Basic that the variable is going to be a number (integer). Well meet alternatives to Integer later.

Number1 = 3

The equals sign is not actually an equals sign. The = sign means assign a value of. In other words, here is where you put something in your variable. We're telling Visual Basic to assign a value of 3 to the variable called number1. Think back to the piece of paper going into the cardboard box. Well, this is the programming equivalent of writing a value on a piece of paper

Now that you have a basic idea of what variables are, let's write a little piece of code to test them out. First, though, let's have our first look at the coding window.

To make life easier, we're going to put a command button on our form. When our command button is clicked, a little message box will pop up. Fortunately, there's no coding to write for a command button, and very little at all for a message box.

Adding a Button to a Form

Add a Button to your VB .NET Forms

Instead of double clicking the Button tool in the toolbox to add the control to the form, we'll explore another way to do it.

With your Form displayed in the Visual Basic Design environment, do the following:

- Click on the Button tool in the toolbox with the left hand mouse button, but click only once
- Move your mouse to a blank area of your form - the mouse pointer will turn into a cross
- Press and hold down the left mouse button
- Drag across the form with the button held down
- Let go of the mouse button when you're happy with the size
- A Button is drawn

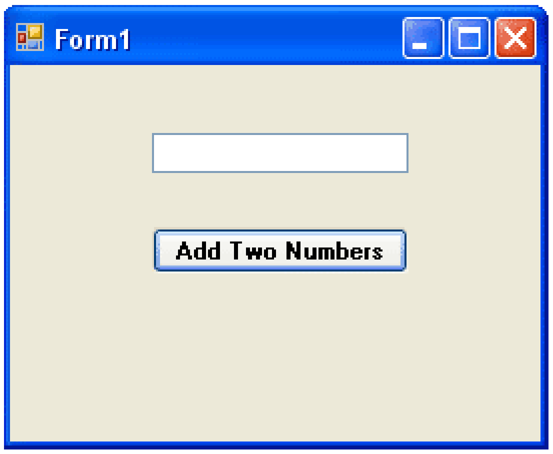
You can use the above method to draw most of the controls onto the form - labels, Buttons, textboxes, etc.

The Button control, just like all the other controls we've seen so far, has a list of properties. One of these properties is the Text property. At the moment, your button will say "Button 1". You can change that to anything you like.

- Click on the Button to highlight it
- Click on Text in the Property Box
- Click in the box next to the word "Text"
- Delete the word "Button 1"
- Type "Add two numbers"
- Click back on the Form

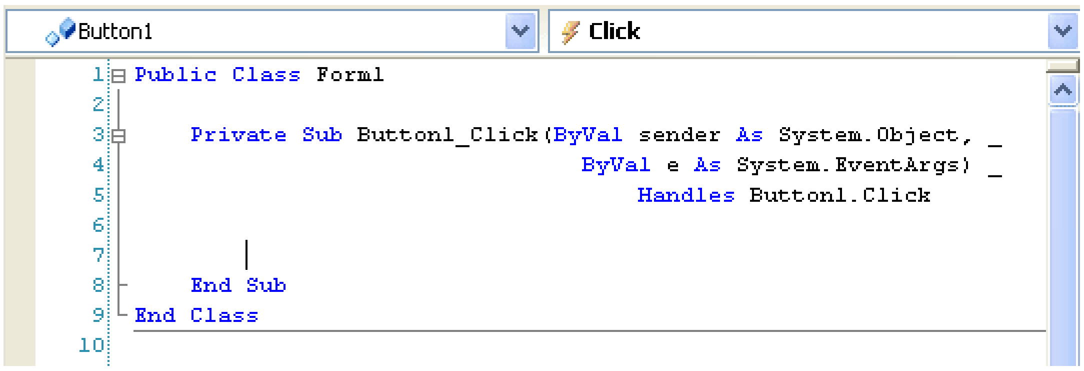
Now add a Textbox to your form using one of the methods outlined (either double-click, or draw).

Your Form should now look something like this:



The Font property of the Button has also been changed, here, in exactly the same way as we changed the Font property of the Label and Textbox previously. The Text for the Textbox control has had its default Text (Textbox 1) deleted.

To get our first look at the code window, double click your Button control. The code window will appear, and will look like this:



```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, _
                              ByVal e As System.EventArgs) _
                                  Handles Button1.Click


    End Sub
End Class

```

Notice that we've used the underscore character (_) to spread the code over more than one line. You can do this in your own code, too, if it becomes to long. But you don't have to.

No more reading these lessons online - get the eBook here!

The part to concentrate on for the moment is where your cursor is flashing on and off. Because you double-clicked the Button control, the cursor will be flashing between the lines **Private Sub** … and **End Sub**.

Here's the part we're concentrating on:

Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) _
Handles Button1.Click

End Sub

The part of the code we're interested in is highlighted in red in the code above. Notice, too, that the underscore character (_) has been used to spread the code over more than one line. You can do this in your own code, too, if it becomes to long:

Private

Private means that no other part of the programme can see this code except for our button

Sub

Short for Subroutine. The "Sub" word tells VB that some code follows, and that it needs to be executed

Button1

This is the name of our button. You might think that we've just erased the word "Button1" when we changed the Text property, so why does VB insist that it's still called Button1? We'll, the Name property of the control is the important one. If you change the Name property, VB will change this button name for you

_Click ( )

This is something called an Event. In other words, when the button is clicked, the Click Event will fire, and the code we're going to write will be executed

End Sub

The subroutine ends right here. This signifies the end of our code

Don't worry if you don't understand all of that. It will become clearer later. Let's add our code, which we'll do on the next page.

Writing your first .NET code

The Text Property of a Control

In the previous section, you just designed a form and had your first look at the code window. We'll add some code right now.

Click your mouse on the blank line after **Private Sub Button1_Click**, etc, but before End Sub. Type the following code:

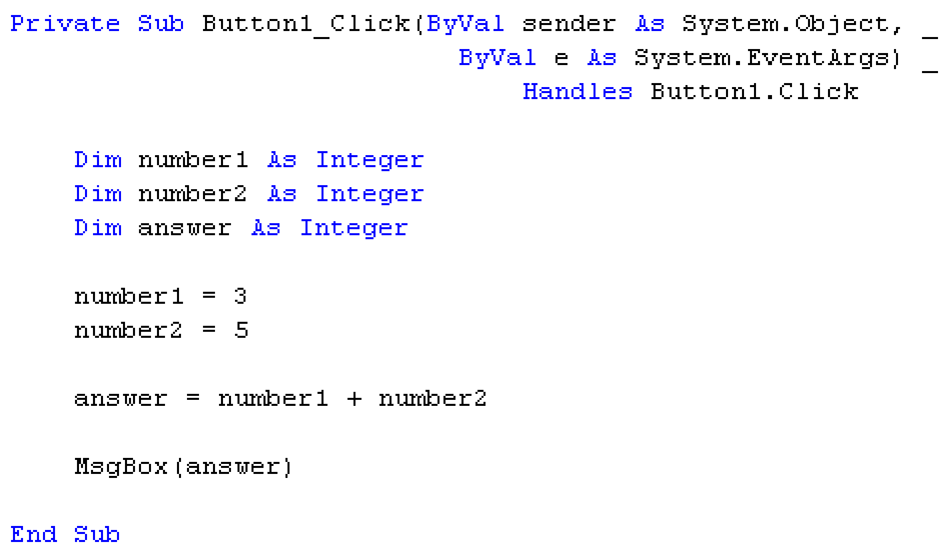
Dim number1 As Integer
Dim number 2 As Integer
Dim answer As Integer

number1 = 3
number2 = 5

answer = number1 + number2

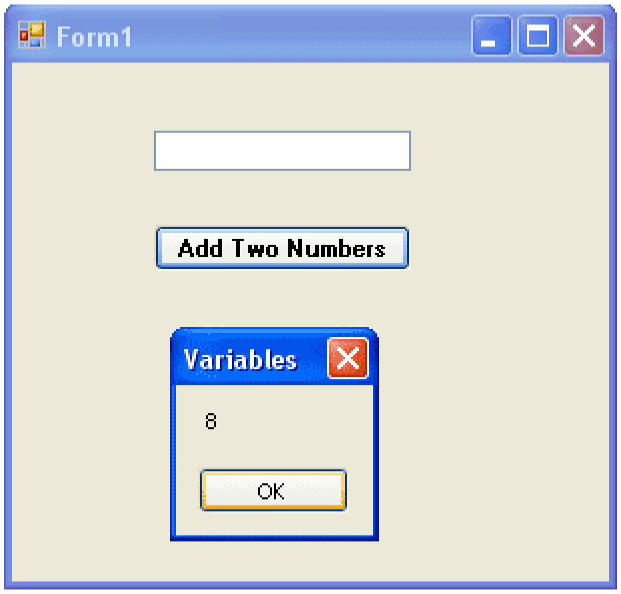
MsgBox answer

After typing all that, your code window should now look like this:

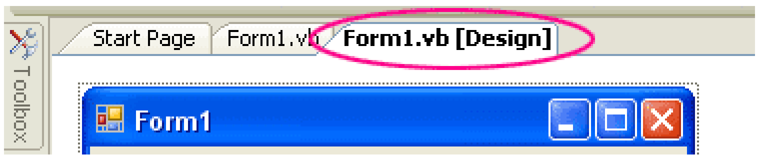


```
Private Sub Button1_Click(ByVal sender As System.Object, _
                                  ByVal e As System.EventArgs) _
                                     Handles Button1.Click

    Dim number1 As Integer
    Dim number2 As Integer
    Dim answer As Integer

    number1 = 3
    number2 = 5

    answer = number1 + number2

    MsgBox(answer)

End Sub
```

Before we explore what's happening here, save your work and then click **Debug > Start** from the Visual Basic Menu, or press F5 on your keyboard. This will launch your programme. Click the Button once, and you should get the following:



Stop your programming, and return to the Design Environment. If you can't see your code, you can click the Tabs at the top of the Window, as in the image below:



Click the "Form1.vb [Design]" tab to see your Form.

OK, what happened there? Well, what happened is we've just wrote a programme to add two numbers together, and we displayed the result by using a Message Box - you very first real programme! But let's break that code down a little more.

- First, we started with the Dim word, indicating to Visual Basic that we wanted to set up a variable
- Then we gave the variable a name (number1)
- Next, we "told" VB that what is going inside the variable is a number (As Integer)
- Two more variable were set up in the same way, number2 and answer

After setting up the three variables, here's what we did:

- Told Visual Basic that what is going into the first variable was the number 3, and what is going into the second variable was the number 5. To put something into a variable, you use the equals (=) sign. But it's not really an equals sign - it's an assignment operator. You are assigning the value of 3 to the variable called number1

number1 = 3
number2 = 5

The next part is a little bit more complicated, but not too complicated. What we wanted to do was to add two numbers together. So we said

number1 + number2

Visual Basic already knows how to add up: all we need to do is "tell" it to add up. We do the "telling" in the traditional, mathematical way - with the plus sign (+). What Visual Basic will do is to look at what we've stored inside number1, and look at what's inside number2. It's sees the 3, sees the five, and also sees the plus sign. Then Visual basic adds them up for you.

Except we also did something else. We said to Visual Basic "When you've finished adding up the two variables number1 and number2, store the result in that other variable we set up, which is called answer."

So, the whole line

answer = number1 + number2

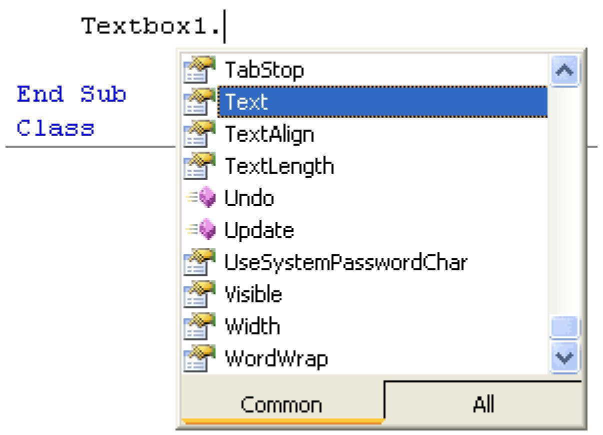
means: "Add the variable called number1 to the variable called number2. Then store the result in the variable called answer."

Think of it as working from the right-hand side of the equals sign first. Then when you have the answer, assign it the variable on the left of the equals sign.

The final part of the programme used Visual Basic's in-built Message Box. We'll learn a more about the Message Box later. For now, think of it as a handy way to display results.

Message boxes are quite handy when you want to display the result of some code. But we have a textbox on the form, and we might as well use that.

So delete the line: MsgBox answer. Type the word Textbox1, then type a full stop. You should see a drop-down box appear. This is a list of the Properties and Methods that the Textbox can use.



```
Textbox1.
End Sub
Class
```

Scroll down until you see the word "Text". Double click the Text property and the drop-down box will disappear. (This drop-down box is known as IntelliSense, and is very handy. It means you can just select a property or method from the list without having to type anything.)

The Text property you have chosen is the same Text property that you set from the Properties Window earlier. Here, we're setting the property with our code; before, we set it at design time. But the result is the same - the Text property of the textbox will be set to a value of our choosing.

To set a value, type an equals sign, then type a value for the Text property. We want the contents of the variable called answer to appear in the textbox. So the rest of the code is just this:

Textbox1.Text = answer

Your code window should then look like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
                            ByVal e As System.EventArgs) _
                                Handles Button1.Click

    Dim number1 As Integer
    Dim number2 As Integer
    Dim answer As Integer

    number1 = 3
    number2 = 5

    answer = number1 + number2

    TextBox1.Text = answer

End Sub
```

Run your code again, and press the Button on the form. You should see the number 8 appear in the textbox.

OK, time for your first exercises. They're not too painful, and hopefully they'll giver you a better idea of what variables are. And besides, programming is about doing, not talking. So off we go!

Exercise

Delete the values "3" and "5" and replace them with numbers of your own

Exercise

Delete the plus sign in between number1 and number2, and replace them with each of the following in turn

- (the minus sign)
* (the multiplication sign in VB is the asterisk sign)
/ (the divide sign in VB is the forward slash)

Exercise

Set up another Integer variable. Give it the name number3. Assign a value of 10 to this new variable. Multiply the value of your new variable by the variable called answer. Display the result in your textbox.

(Another way to assign values to variables is when you first set them up. You can do this:

Dim number3 As Integer = 10

This is exactly the same as saying:

```
Dim number3 As Integer

number3 = 10
```

It's up you which method you use. But the objective is the same - to assign a value to a variable.)

In the next part, we'll about a different kind of variable - a string variable.

String Variables

An Introduction to Programming Strings

So we've learnt something about variables, what they are and how to set one up. We learnt about the word "integer", and that integer variables held numbers. But what if we don't want numbers? After all, our first Form asked users to type in their First Name and Last Name. Names are not numbers, so what do we do then? Well that's where Strings come in.

What is a String? Actually a string is nothing more than text. And if we want Visual Basic to store text we need to use the word "String". To set up a variable to hold text we need to use As String and not As Integer. If the information we want to store in our variables is a First Name and a Last Name, we can set up two variables like this.

```
Dim FirstName As String
Dim LastName As String
```

Again we've started with the Dim word. Then we've called the first variable FirstName. Finally, we've ended the line by telling Visual Basic that we want to store text in the variable - As String.

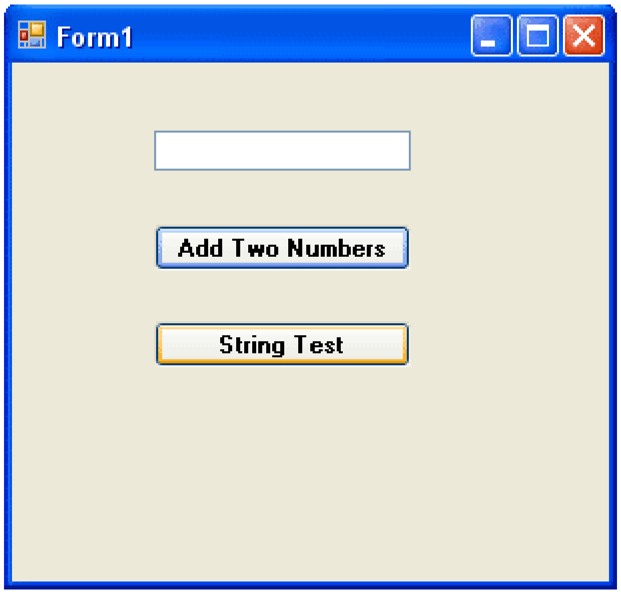
So we've set up the variables. But there is nothing stored in them yet. We store something in a variable with the equals sign (=). Let's store a first name and a last name in them

```
FirstName = "Bill"
LastName = "Gates"
```

Here, we said to Visual Basic "Store the word 'Bill' into the variable FirstName and store the word 'Gates' into the variable called LastName. But pay attention to the quotation marks surrounding the two words. We didn't say Bill, we said "Bill". Visual Basic needs the two double quotation marks before it can identify your text, your String.

So remember: if you're storing text in a variable, don't forget the quotation marks!

To test all this out, add a new Button to your Form. Set the Text property of the Button to "String Test". Your Form would then look like this:



Double click your new button, and add the following code:

```
Dim FirstName As String
Dim LastName As String
Dim FullName As String

FirstName = "Bill"
LastName = "Gates"

FullName = FirstName & LastName

Textbox1.Text = FullName
```

Your code window should now look like this (some of the first line has been cropped in the image below):

```
Private Sub Button2_Click(ByVal sender As System.Object, _
                           ByVal e As System.EventArgs) _
                               Handles Button2.Click

    Dim FirstName As String
    Dim LastName As String
    Dim FullName As String

    FirstName = "Bill"
    LastName = "Gates"

    FullName = FirstName & LastName

    TextBox1.Text = FullName

End Sub
```

There's a line there that needs explaining

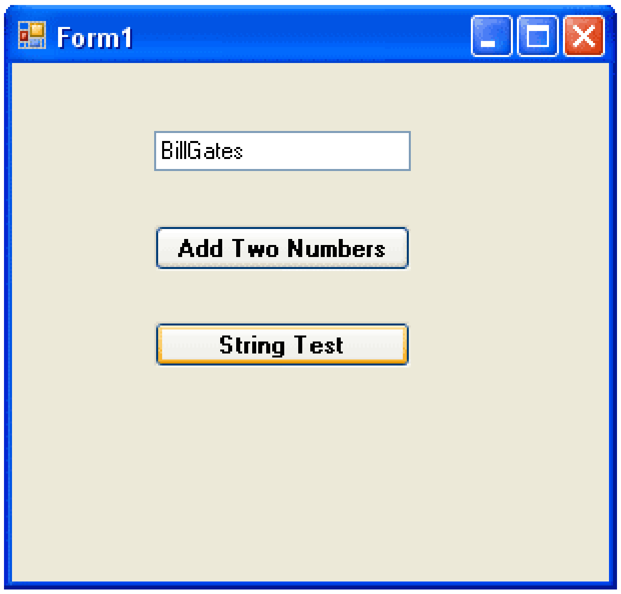
FullName = FirstName & LastName

In the two lines of code above that one, we stored the string "Bill" and the string "Gates" into two variables. What we're doing now is joining those two variables together. We do this with the ampersand symbol (&). The ampersand is used to join strings together. It's called Concatenation.

Once Visual Basic has joined the two strings together (or concatenated them), we're saying "store the result in the variable called FullName". After that, we tell VB to display the result in our Textbox.

So, once you've typed the code, start your programme and test it out.

Once the programme is running, Click the Button and see what happens. You should have a Form that looks something like this one:



The textbox displays the text stored in our variables, "Bill" and "Gates". We joined them together with the ampersand (&). But as you can see, the two words are actually joined as one. We can add a bit of space between the two words by using another ampersand. Change this line **FullName = FirstName & LastName** to this:

FullName = FirstName & " " & LastName

What we're saying here is join this lot together: the variable called FirstName and a single blank space and the variable called LastName. When you've finished concatenating it all, store the result in the variable FullName.

Notice that we don't surround FirstName and LastName with quotation marks. This is because these two are already string variables; we stored "Bill" into FirstName and "Gates" LastName. So VB already knows that they are text.

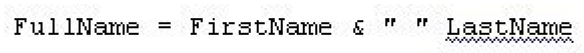
No more reading these lessons online - get the eBook here!

Exercise

Remove one of the ampersand symbols (&) from this line in your code:

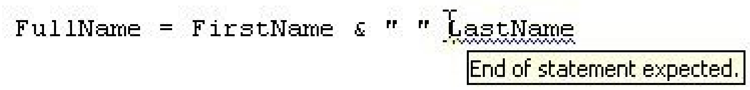
FullName = FirstName & " " & LastName

Move your cursor down a line or two. You should see that part of your code has a wiggly blue line under it:



```
FullName = FirstName & " " LastName
```

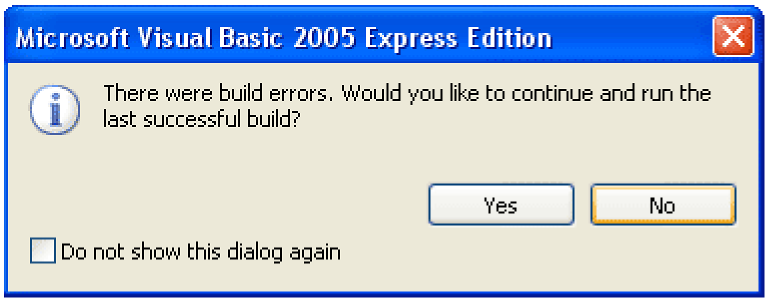
VB is telling you that it has problems with this line of code. If you hold your mouse over the wiggly blue line, VB tries to provide an explanation:



```
FullName = FirstName & " " LastName
```

End of statement expected.

The explanations VB provides are sometimes enigmatic. But you will know that there is a problem. If you run the code, you'll get this popping up at you:



Microsoft Visual Basic 2005 Express Edition

There were build errors. Would you like to continue and run the last successful build?

Click the NO button. Put the ampersand back in, and all will be well.

Exercise

Amend your code so that the textbox reads Gates Bill when the Command button is clicked.

Exercise

Add another string variable to your code. The variable should hold a middle name. Display the first name, the middle name and the last name in the textbox.

Points to remember:

- Your variable names cannot include spaces. So the variable **MiddleName** would be all right, but **Middle Name** will get you an error message
- When you're putting text into your new variable, don't forget the two double quotes
- Remember to put in enough ampersands in your **FullName =** line of code

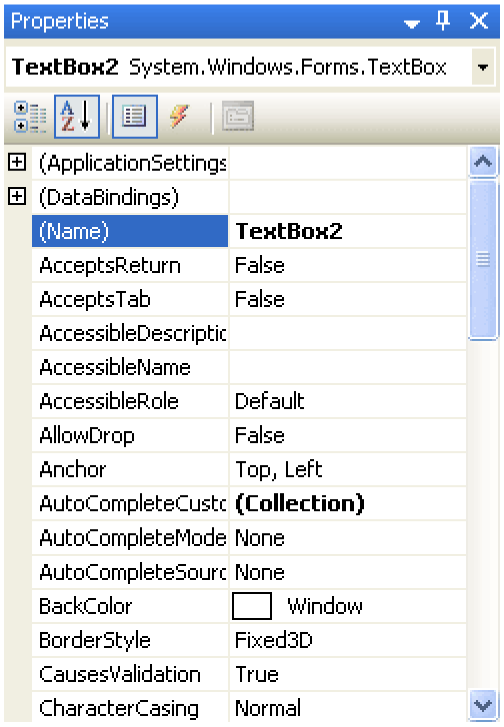
In the next part, we'll take a look at how to asign text from a textbox into our string variables.

Assigning Textbox text to your Variables

Working with Textboxes

Instead putting direct text into your variables, such as "Bill" or "Gates", you can get text from a textbox and put that straight into your variables. We'll see how that's done now. First, do this:

- Add a new textbox to your form
- With the textbox selected, locate the **Name** property in the Properties area:



TextBox2 System.Windows.Forms.TextBox

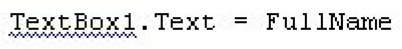
(ApplicationSettings	
(DataBindings)	
(Name)	TextBox2
AcceptsReturn	False
AcceptsTab	False
AccessibleDescriptic	
AccessibleName	
AccessibleRole	Default
AllowDrop	False
Anchor	Top, Left
AutoCompleteCustc	(Collection)
AutoCompleteMode	None
AutoCompleteSourc	None
BackColor	⬜ Window
BorderStyle	Fixed3D
CausesValidation	True
CharacterCasing	Normal

The current value of the Name property is **Textbox2**. This is not terribly descriptive. Delete this name and enter **txtLastName**. Scroll down and locate the Text property. Delete the default text, and just leave it blank.

Click on your first textbox to select it. Change the Name property from **Textbox1** to **txtFirstName**.

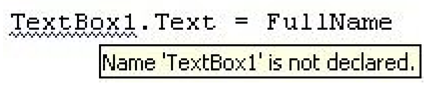
What we've done is to give the two textboxes more descriptive names. This will help us to remember what is meant to go in them.

Unfortunately, if you view your code (click the Form1.vb tab at the top, or press F7 on your keyboard), you'll see that the blue wiggly lines have returned:



```
TextBox1.Text = FullName
```

If you hold your cursor of the Textbox1, you'll see this:



```
TextBox1.Text = FullName
Name 'TextBox1' is not declared.
```

It's displaying this message because you changed the name of your Textbox1. You now no longer have a textbox with this name. In the code above, change Textbox1 into **txtFirstName** and the wiggly lines will go away. (Change it in your Button1 code as well.) Your code should now read:

txtFirstName.Text = FullName

Run your programme again. If you see any error messages, stop the programme and look for the wiggly lines in your code.

We'll now change our code slightly, and make use of the second textbox. You'll see how to get at the text that a user enters.

Locate these two lines of code

FirstName = "Bill"
LastName = "Gates"

Change them to this

FirstName = txtFirstName.Text
LastName = txtLastName.Text

Remember: the equals (=) sign assigns things: Whatever is on the right of the equals sign gets assigned to whatever is on the left. What we're doing now is assigning the text from the textboxes directly into the two variables.

Amend your code slightly so that the Whole Name is now displayed in a message box. Your code should now be this:

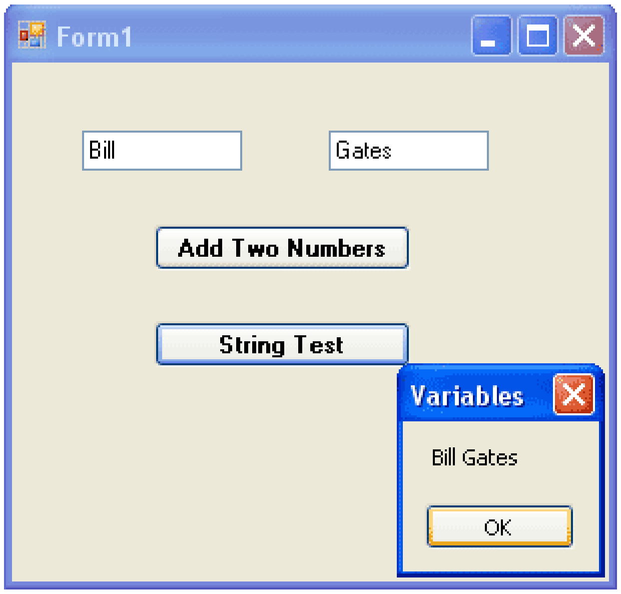
Dim FirstName As String
Dim LastName As String
Dim WholeName As String

FirstName = txtFirstName.Text
LastName = txtLastName.Text

WholeName = FirstName & " " & LastName

MsgBox(WholeName)

Run your programme. Enter "Bill" in the first textbox, and "Gates" in the second textbox. Then click your "String Test" button. You should get this:



Bill Gates

Before we changed the code, we were putting a person's name straight in to the variable FirstName

FirstName = "Bill"

But what we really want to do is get a person's name directly from the textbox. This will make life a whole lot easier for us. After all, not everybody is called Bill Gates! In the line **FirstName = txtFirstName.Text** that is what we're doing - getting the name directly from the textbox. What we're saying to Visual Basic is this

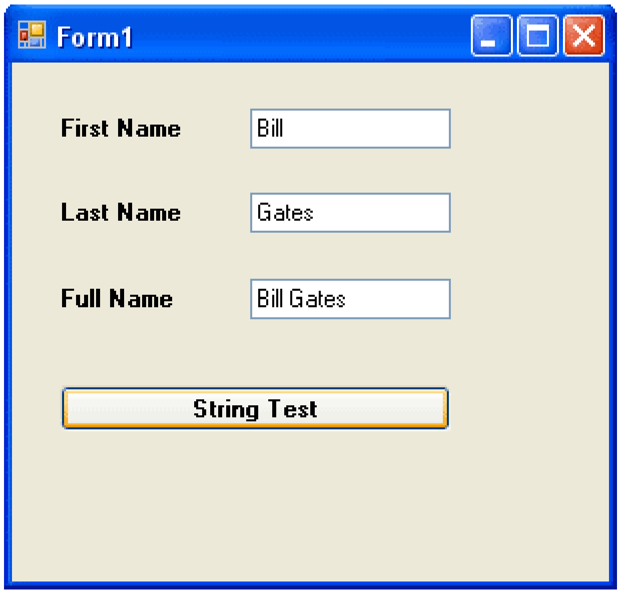
- Look for a Textbox that has the Name txtFirstName
- Locate the Text property of the Textbox that has the Name txtFirstName
- Read whatever this Text property is
- Put this Text property into the variable FirstName

And that's all there is too reading values from a textbox - just access its Text property, and then pop it into a variable.

Exercise

- Add a third textbox to your form
- Change its Name property to txtWholeName
- Add labels to your form identifying each textbox (A quick way to add more labels is to use the toolbox to add one label. Then right click on that label. Choose Copy from the menu. Right click on the form, and select Paste.)
- Write code so that when the "String Test" button is clicked, the whole of the persons name is displayed in your new textbox

When you complete this exercise, your form should look like this one (we've deleted the first button and its code, but you don't have to):



In the next part, we'll explore some more variable types you can use.

More about Variables in VB NET

VB NET Variables

We've met two variable types so far - **As String** and **As Integer**. But there are quite a few more you can use. Let's start by examining number variables.

Start a new project for this. If you have the old one displayed, you can click **File > Close Solution** from the menu bar. You will then be returned to the Start Page. Click the New **Project** button at the bottom. In the dialogue box, give your project a name.

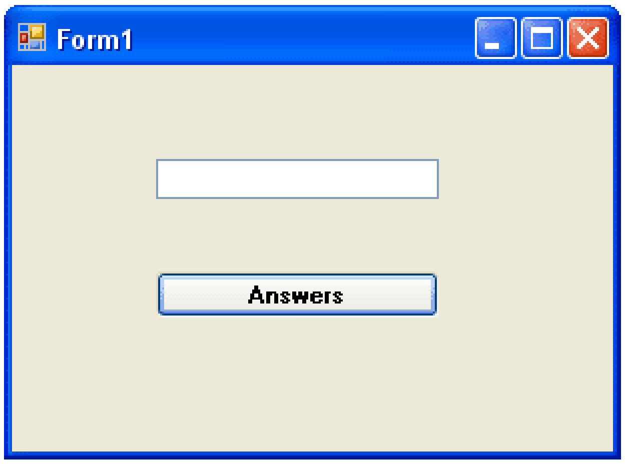
Put a textbox and a Button on your new form. Change the Properties of the Textbox to the following

Name: txtNumbers
Font: MS Sans Serif, Bold, 10
Text: just delete the default Textbox1, and leave the textbox blank

Change the Properties of the Button to the following:

Text: Answers
Font: MS Sans Serif, Bold, 10

Click on the Form itself, and change it's **Text** property to "Testing Types". Your Form should look something like this:



Double click on the Button to bring up the code window. Type the following code for your Button (The **Button1_Click** part is spread over three lines only for ease-of-reading on this web page. You can keep yours on one line in your code):

```
Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) _
Handles Button1.Click
```

Dim testNumber As Short

testNumber = txtNumbers.Text

MsgBox testNumber

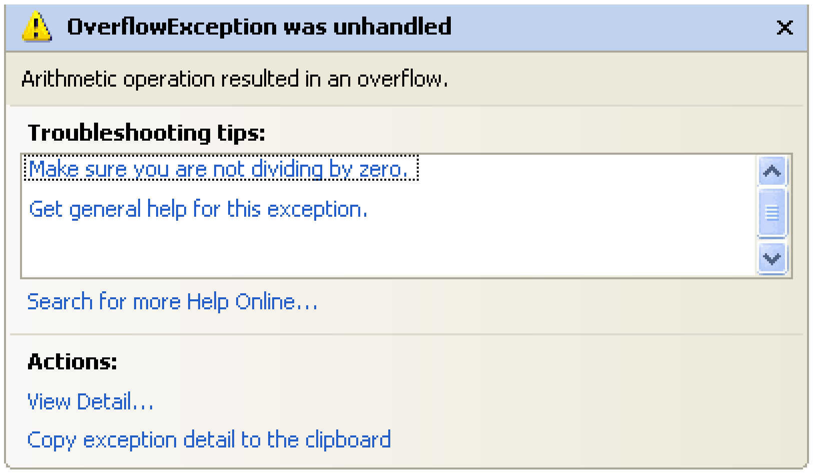
End Sub

Notice that there is a new Type of variable declared - As Short. This means "Short Integer". We'll see what it does.

Run your programme. While it's running, do the following:

- Enter the number 1 into the textbox, and click the Answers button
- The number 1 should display in the Message Box
- Add the number 2 to the textbox and click the Button
- The number 12 should display in the Message Box
- Add the number 3 to the textbox and click the Button
- The number 123 should display in the Message Box
- Keeping adding numbers one at a time, then clicking the button

How many numbers did you get in the textbox before the following error message was displayed? (Click Break to get rid of it.)



OverflowException was unhandled

Arithmetic operation resulted in an overflow.

Troubleshooting tips:

Make sure you are not dividing by zero.

Get general help for this exception.

Search for more Help Online...

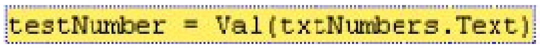
Actions:

View Detail...

Copy exception detail to the clipboard

You should have been able to enter **12345** quite safely. When you entered **123456** and clicked the button, that's when the error message displayed.

When you click the Break button, you are returned to the coding environment. You'll see the problem line highlighted in yellow:



```
testNumber = Val(txtNumbers.Text)
```

But your programme will still be running. So click **Debug > Stop Debugging** to return to the normal code window.

An Overflow happens when you try to put too much information into a variable that can't handle it.

No more reading these lessons online - get the eBook here!

The reason we got an error message after just 6 numbers was because of the variable type. We had this

Dim testNumber As Short

And it's As Short that is causing us the problems. If you use As Short you're only allowed numbers up to a certain value. The range for a Short variable is -32 768 to 32 767. When we entered 6 numbers, Visual Basic decided it didn't want to know. If you run your programme again, and then enter 32768, you'll get the same Overflow error message. If you change it once more to -32769, you'll get the error message as well. So it's not just 6 numbers a Short Type can't handle - it's 5 numbers above or below the values specified.

So what's the solution? Change the variable Type, of course!

Change the variable to this

Dim testNumber As Integer

Now start the programme and try again, adding numbers one at a time to the textbox, and then clicking the Command button. How far did you get this time?

If you started at 1 and added the numbers in sequence, you should have been allowed to enter 1234567890. One more number and Visual Basic gave you the Overflow error message, right? That's because variable types with As Integer also have a limitation. The range you can use with the As Integer variable type is -2,147,483,648 to 2,147,483,647. If you want a really, really big number you can use As Long.

Dim testNumber As Long

But these will get you whole numbers. Leave your number on As Integer. Run your programme again and enter a value of 123.45 into your textbox. Press the button and see what happens.

VB will chop off the point 45 bit at the end. If you want to work with floating point numbers (the .45 bit), there are three Types you can use:

Dim testNumber As Single
Dim testNumber As Double
Dim testNumber As Decimal

Single and Double mean Single-Precision and Double-Precision numbers. If you want to do scientific calculations, and you need to be really precise, then use Double rather than Single: it's more accurate.

The **As Decimal** Type is useful when you want a precise number of decimal places. It's not as accurate as the Double Type, though.

In terms of the space used in the computer's memory, Short Types use 2 Bytes, Integer Types use 4 Bytes, Long Types use 8 Bytes, Single Types use 4 Bytes, Double Types use 8 Bytes, and Decimal Types use 16 Bytes.

Exercise

Write a programme to calculate the following sum.

0.123345678 * 1234

Use the Single Type first, then change it to **As Double**. Use a Message box to display the answer. Was the number rounded up or rounded down for the Single Type?

In the next part, we'll get some more practise with variables.

Using Variables in your NET Code

How to Use Variables in VB NET

In this next section, we're going to learn how to transfer the contents of one textbox to another textbox. We'll also learn to transfer the text from a label to a textbox, and whatever was in the textbox we'll transfer it to a label. This will get us a little more practise with variables, and how to use them.

Ok, start a new Visual basic project. You should know how to do this by now, and what the design environment looks like. But you should have a plain grey Form on your screen. By default it will be called Form1.

Make sure the Form is selected (has it got the white squares around it?), and the click the **Name** property in the Properties window. Change the Name of the form to **frmVariables**.

Set the Text property of the Form to "Transferring information". You can choose any background colour you like for the form, or leave it on the default.

Put the following controls on the Form, and change their properties to the one's specified below (NOTE: **lbl** is short for label):

Textbox

Name: txtVariables
Font: MS Sans Serif, Bold, 10
Text Delete the default text "Text1" and leave it blank

Label

Name: lblTransfer
BackColor: A colour of your choice
Text: Label Caption
Font: MS Sans Serif, Bold, 10

Button

Name: btnTransfer
Text: Transfer

The height of your controls is entirely up to you.

If you double click your Button to bring up the code window, you will see that the first line of the code no longer says Button1_Click (etc). The first line should say this

```
Private Sub btnTransfer_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) _
Handles btnTransfer.Click

End Sub
```

The reason it has changed is because you changed the Name property of the Button. The button now has the Name btnTransfer. If you wanted to, you could change the Name property back to Button1. Then when you double clicked the button, the code window would pop up and the first line would be Button1_Click(etc).

What we're going to do now is to transfer the Text on the label ("Label Caption") to our empty textbox. And all with the click of a button.

As you'll see, there isn't much code.

Type the following into your code window:

Dim LabelContents As String

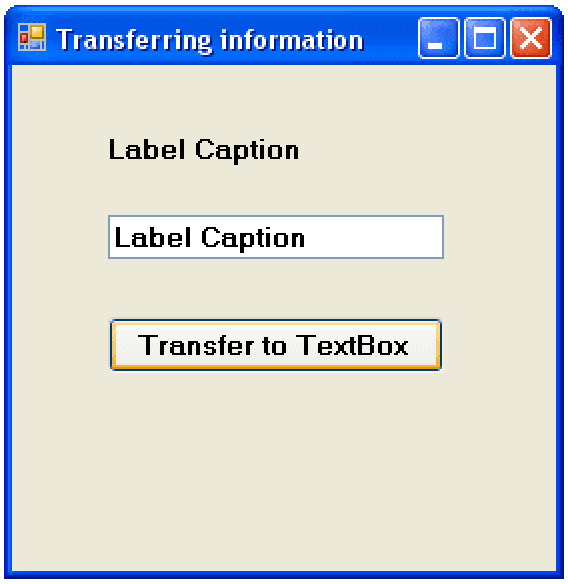
LabelContents = lblTransfer.Text

txtVariables.Text = LabelContents

Your code window should now look something like this:

```
Private Sub btnTransfer_Click(ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
                                    Handles btnTransfer.Click

    Dim LabelContents As String

    LabelContents = lblTransfer.Text

    txtVariables.Text = LabelContents

End Sub
```

Now Run your programme and test it out. When you click on the "Transfer" button, you should see that the contents of the label will be inserted into the textbox:



But let's break the code down and see what's going on.

No more reading these lessons online - get the eBook here!

Dim LabelContents As String

Here is where we set up a variable called LabelContents. Because it will be holding text, we've used the variable type As String.

LabelContents = lblTransfer.Text

Here is where we put something into our empty variable. We changed the Name property of our Label from the default Label1 to lblTransfer. A Label has lots of properties you can manipulate. One of those properties is the Text property. After you typed the word "lblTransfer" and then typed a full stop, you probably saw a drop down box appear. Inside the box is a list of all the properties and Methods that a Label has. We wanted to manipulate the Text property of our label so we selected the word Text after the full stop. So we were saying "Access the value of the Text property of the label called lblTransfer, and put this value into the variable called LabelContents." Because our Text was ""Label Caption", the variable LabelContents now holds the text "Label Caption."

txtVariables.Text = LabelContents

Finally, we want to transfer whatever is in the variable LabelContents to the Textbox. Our Textbox is called txtVariables. Again, after typing the full stop the drop down box would appear, showing you a list of all the properties that a Textbox has. The one we're interested in is the Text Property. So we're saying, "Take whatever text is in the variable LabelContents, and transfer it to the Text property of the Textbox called txtVariables.

And with three lines of code we can transfer text from a label to a Textbox. But can we do it the other way round? Can we transfer whatever is in a Textbox to a Label? Well, sure we can.

Add another button to your form. Change its Name property from Button1 to cmdTransferToLabel, and change the Caption property to "Transfer To Label". Again, there's just three lines of code.

So double click your new button to bring up the code window. Then type in the following code:

```
Dim TextBoxContents As String

TextBoxContents = txtVariables.Text

lblTransfer.Text = TextBoxContents
```

Now, see if you can work out how it works. It's the same thing as the first three lines of code: set up a variable, transfer the Text property of the Textbox to the variable, transfer the variable to the Text property of the Label. Run your programme and test it out. Type something into the Textbox, and then click the "Transfer To Label" button.

Exercise

A button also has a Text Property. Write code to transfer the Text property of a button to the Textbox. It's probably better for this exercise to create a new Button. Set its Name property to whatever you like. And give its Text Property a new value (The Text property will be Button1 by default) .

But the process is exactly the same as the two bits of code above - you should only need 3 lines of code for this exercise.

- Set up a Variable
- Transfer the Text property of the button to the variable
- Transfer the variable to the Textbox

In the next part, we'll start a Calculator project. This will get you some more practical experience of working with variables.

A Calculator Project in VB NET

VB NET Calculator

In the next few pages, you're going to create a Calculator. It won't be a very sophisticated calculator, and the only thing it can do is add up. What the project will give you is more confidence in using variables, and shifting values from one control to another. So create a new project, call it **Calculator**, and let's get started.

Designing the Form

Let's design the form first. What does a calculator need? Well numbers, for one. A display area for the result. A plus sign button, an equals sign button, and a clear the display button.

Here's how our calculator is going to work. We'll have 10 button for the numbers 0 to 9. When a button is clicked its value will be transferred to a display area, which will be a Textbox. Once a number is transferred to the Textbox we can click on the Plus button. Then we need to click back on another number. To get the answer, we'll click on the equals sign. To clear the display, we'll have a Clear button.

If you haven't already, create a new project. Save it as Calculator. To your new form, first add ten Buttons (You can add one, then copy and paste the rest). The Buttons should have the following Properties:

Name: ***btn*** Plus a Number (btnOne, btnTwo, btnThree, etc)

Text: A number from 0 to 9. A different one for each button, obviously

Font: MS Sans Serif, Bold, 14

Next, add a Textbox. Set the following properties for the Textbox:

Textbox
Name: txtDisplay
Font: MS Sans Serif, Bold, 14
Text: Erase the default, Textbox1, and leave it blank

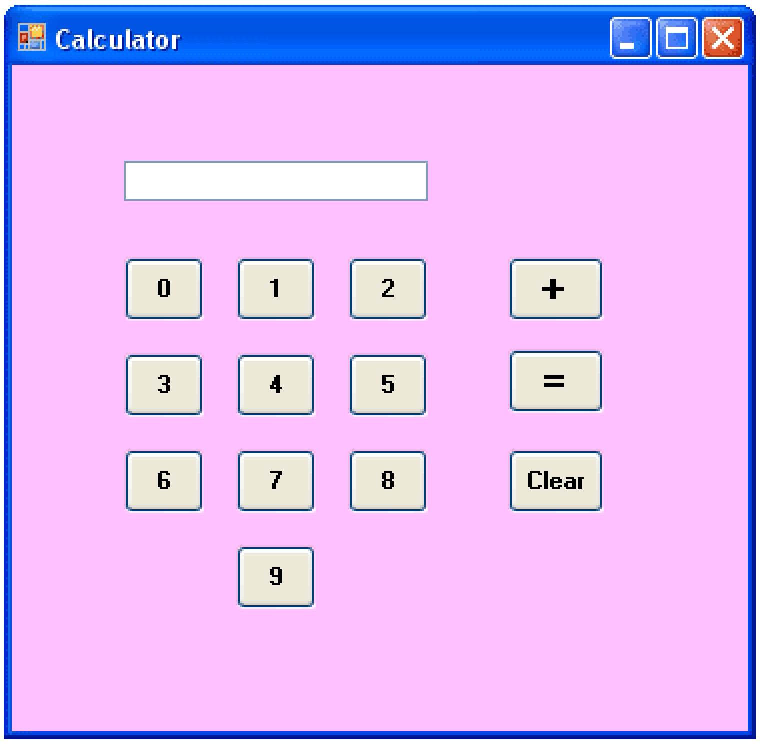
Three more Command buttons need to be added

Plus Button
Name cmdPlus
Font MS Sans Serif, Bold, 14
Text +

Equals Button
Name cmdEquals
Font MS Sans Serif, Bold, 14
Text =

Clear Button
Name cmdClear
Font MS Sans Serif, Bold, 14
Text Clear

When your form design is finished, it might look something like this:



So if you wanted to add 5 + 9, you would click first on the 5. A 5 would appear in the textbox. Then you would click the + symbol. The 5 would disappear from the textbox. Next, click on the 9. The number 9 would appear in the textbox. Finally, click on the = symbol. The 9 would disappear from the textbox, and the answer to our sum would replace it. We would then click the Clear button to clear the display.

In the next section, we'll make a start on the VB NET code for the all those buttons.

The Code for the VB NET Calculator

VB NET Calculator - the Code

In the previous part, you designed the NET form for your calculator. We'll now make a start on the code.

You might be thinking that all this is terribly complicated at such an early stage. But it isn't really. All we are doing is transferring the Text Properties from the Buttons to the textbox. And you already know how to do that. The number buttons don't do anything else. All the work is done with the Plus button and the Equals buttons. And there's only two lines of code needed for the Plus button, and three for the Equals button.

For this to work, though, a little word about Scope in VB NET.

So far, when you've set up a variable, you've set them up behind a Private Subroutine. Like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) _
Handles btnZero.Click

Dim MyVariable As String

End Sub
```

Suppose you had another button on the form, Button2, and the code was this

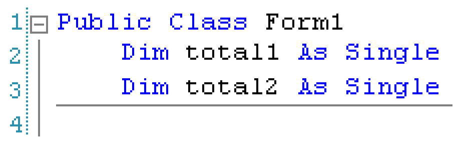
```
Private Sub Button2_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) _
Handles btnZero.Click

Dim MyOtherVariable As String

End Sub
```

How can you access what's in **MyVariable** from Button2? The answer is, you can't. It's like two people sitting at desks in cubicles. Each person has written something on a piece of paper. They can't see into the other person's cubicle, only whatever is their own cubicle. So how do they share their information?

Well suppose there is a screen in front of them. A big screen. They can both see the screen in front of them; it's each other they can't see. What they could do is project their information onto the screen. Then one person could see what the other has written.

Similarly, in VB you can set up your variable declarations outside of the code for a Button. That way, more than one Button can see the code.

You can place your variable declarations right at the top of the code window, just beneath the line that begins "Public Class Form1". We'll set up two Integer variables there, **total1** and **total2**:



```
Public Class Form1
    Dim total1 As Single
    Dim total2 As Single
```

Now all the controls on your form can see these two variables. Those Buttons you set up can put something in them, and every button has the ability to see what's inside them.

The 0 to 9 Bu ons

The Buttons with the Text 0 to 9 only need to do one thing when the button is clicked - have their Text Properties transferred to the Textbox. You've already wrote code to do that.

So double click the 0 Button and enter the following code:

```
Private Sub btnZero_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) _
Handles btnZero.Click

txtDisplay.Text = btnZero.Text

End Sub
```

This code will transfer the Text Property of a Button called btnZero to the Text Property of a Textbox called txtDisplay.

Run your programme and try it out. When the programme is running, click the 0 button to see that it does indeed transfer the Text on the Button to the textbox

Except, there's a problem with that code. If you wrote similar code for all ten of your number buttons, the calculator wouldn't be right. Why is that? Have you spotted what's wrong? It's a good idea to set this book aside for a while and think about why this code on it's own wouldn't work. In fact you could write code for a few more of the number buttons and test it out.

What happens when you transfer the number 2 to the Textbox, and then click the number 3? The number 2 will disappear, to be replaced by the number 3. Which is all right if all you wanted to do was add up single numbers, but not much good if you actually wanted the number 23 in the Textbox. With this code, you could have either the number 2 in the Textbox or the number 3, but not both!

So how do we solve this problem? How do we fix it so that we can have two or more numbers appearing in our Textbox?

What we need is a way to get whatever is in the Textbox to stay where it is, and not disappear on us when we click a different number. It's quite easy. It's this:

txtDisplay.Text = txtDisplay.Text & btnZero.Text

So now we're saying the textbox doesn't just contain the Text on the Button. It must also keep whatever is inside the textbox as well.

So what you need to do now is to add that code to all of your ten number Buttons. Obviously it won't be exactly the same. For the button called btnOne the code would be this:

txtDisplay.Text = txtDisplay.Text & btnOne.Text

When you've finished coding all ten buttons, run the programme and click all ten number button to see if they do indeed transfer the numbers on the caption to the textbox. Not only that, but test to see if you can have more than one number in the textbox.

Now that we can get numbers into our Textbox display area, we'll write code to do something with those numbers - add them together, in other words. We'll do that in the next part. Click the link below to move on.

Coding the Plus Button

VB NET Calculator - Adding up

Let's remind ourselves how our calculator from the previous section works works. To add up 5 + 9, we'd do this:

1. Click first on the 5
2. A 5 appear in the textbox
3. Click the + symbol
4. The 5 disappears from the textbox
5. Click on the number 9
6. A 9 appears in the textbox
7. Click on the = symbol
8. The 9 disappears from the textbox
9. The answer to 5 + 9 appears in the textbox
10. Click the "Clear" button to clear the textbox

We've done numbers 1 and 2 on that list. We're now going to do numbers 3 and 4 on the list. What we're trying to do is this: Click on the Plus symbol and make the number in the Textbox disappear. Before the number vanishes, we can store it in a variable. The variable we're going to be storing the number in is one of those variable we set up at the top of the code. It's this one:

```
Dim total1 As Integer
```

We've already seen how to retain a value from a textbox and add it to something else:

```
txtDisplay.Text = txtDisplay.Text & btnZero.Text
```

Here, we kept the value that was already in the textbox and joined it to the Text property of the button.

We can do something similar if we want to retain a value that is already in a variable. Examine this:

```
variable1 = variable1 + 1
```

The "**= variable1 + 1**" part just says "Remember what is in the variable variable1, and then add 1 to it. So if variable1 contain the number 3, what would variable1 now hold after that bit of code is executed? The whole code might be this

```
variable1 = 3
variable1 = variable1 + 1
```

(If you don't know the answer to that, please send an email and ask for some further clarification on the subject.)

The above is known in programming terms as "Incrementing a variable". There is even a shorthand you can use:

```
variable1 += 1
```

This says "variable1 equals variable1 plus 1". Or "Add 1 to variable1". You can also use the other mathematical operators with the same shorthand notation:

```
variable1 = 10
variable1 *= 3
```

This new code says "Multiply whatever is inside of variable1 by 3".

The shorthand notation can be tricky to read (and to get used to), so we won't use it much. But it's there is you want it.

Back to our code.

If we're going to be adding two numbers together, we need Visual Basic to remember the first number. Because when we click the equals sign, that first number will be added to the second number. So if we haven't asked VB to remember the first number, it won't be able to add up.

The variable we're going to be storing the first number in is total1. We could just say this:

total1 = txtDisplay.Text

Then whatever is in the textbox will get stored in the variable total1.

Except we want VB to remember our numbers. Because we might want to add three or more numbers together: 1 + 2 + 3 + 4. If we don't keep a running total, there's no way for our programme to remember what has gone before, it will just erase whatever is in total1 and then start again.

So just like the code above (varaible1 = variable1 + 1), we need to "tell" our programme to remember what was in the variable. We do it like this:

total1 = total1 + Val(txtDisplay.Text)

That Val( ) part just makes sure that a number in a textbox is kept as a number, and not as text. It's short for Value. The important part is the total1 + txtDisplay.Text. We're saying "The variable total1 contains whatever is in total1 added to the number in the textbox." An example might clear things up. Suppose we were adding 5 + 9. Now, suppose total1 already contained the number 5, and that number 9 was in the textbox. It would work like this:

Finally, we need to erase the number in the textbox. To erase something from a textbox, just set its Text property to a blank string. We do this with two double quotation marks. Like this:

txtDisplay.Text = ""

That tiny bit of code will erase the Text a textbox. Another way to erase text from a textbox is to use the Clear method. After you typed a full stop, you probably saw the drop down list of Properties and Methods. Scroll up to the top, and locate the word Clear. Double click "Clear" and the drop down list will close. Hit the return key and VB adds two round brackets to your code:

txtDisplay.Clear( )

Notice that we're not setting the textbox to equal anything. We're using something called a Method. You can tell it's a Method because there's a purple block icon next to the word. A

Method is a built-in bit of code that VB knows how to execute. In other words, it knows how to clear text from a textbox. You'll learn more about Methods later.

So the whole code for our Button called btnPlus is this:

```
Private Sub btnPlus_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) _
Handles btnPlus.Click

total1 = total1 + Val(txtDisplay.Text)

txtDisplay.Clear()

End Sub
```

Add that code to your Plus button. All we've done with that code is to store numbers into our variable total1 and then erase whatever was in the textbox.

We now need to add the numbers up.

Exercise

Write the code for the equals button. There's only three lines in total, and here's a little help.

You need to use the other variable that was set up at the top of the coding window, total2. The variable total1 will be added to whatever is total2

The first line of code will be this

```
total2 = total1 + (something missing here)
```

Your job is to supply the missing code. In other words, replace "(something missing here)"

Remember that total1 contains the first number to be added. And you know where that came from. The only thing left to do is to add the second number.

For the second line of code, you need to transfer the total2 variable to the textbox.

For the third line of code, all you are doing is resetting the variable total1 to zero. That's because after the equals button has been pressed, we have finished adding up. If you wanted to do some more adding up, that total1 will still hold the value from the last time. If you reset it to zero, you can start adding some new numbers.

The only thing left to do is to write a single line of code for the Clear button. All you are doing there is erasing text from a textbox. Which you already know how to do.

When you're finished, you should have a simple calculator that can add up integers. In the next section, we'll take a look at Conditional Logic, and why it's so important for your programming skills.

Conditional Logic - If Statements

If Statements in VB NET

What is conditional logic? Well, it's something you use in your daily life all the time, without realising you're doing it. Suppose that there is a really delicious cream cake in front of you, just begging to be eaten. But you are on a diet. The cake is clearly asking for it. So what do you do, eat the cake and ruin your diet? Or stick to your diet and let somebody else have that delicious treat? You might even be saying this to yourself:

If I eat the cake **Then** my diet will be ruined

If I don't eat the cake **Then** I will be on course for a slimmer figure

Note the words **If** and **Then** in the above sentences. You're using conditional logic with those two words: "I will eat the cake on condition that my diet is ruined". Conditional logic is all about that little **If** word. You can even add Else to it.

If I eat the cake **Then** my diet will be ruined

Else

If I don't eat the cake **Then** I will be on course for a slimmer figure

And that is what conditional Logic is all about - saying what happens if one condition is met, and what happens if the condition is not met. Visual Basic uses those same words - If, Then, Else for conditional Logic. Let's try it out.

Start a new project. Give it any name you like. In the design environment, add a Button to the new form. Double click the button and add the following code to it:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) _
Handles Button1.Click

Dim firstname As String

firstname = "Bill"
If firstname = "Bill" Then MsgBox("firstname is Bill")

End Sub
```

Run the programme and see what happens. You should get a Message Box with the words "firstname is Bill" in it.

What we did was to set up a string variable and put the name "Bill" into it. When then used conditional logic to test what was in the variable. In fact, we used an If statement. We said:

If the variable called firstname holds the value "Bill" Then display a Message Box

We can tidy that up a bit, because a single line of code can get very long with If statements. We can use this format instead.

```
If firstname = "Bill" Then
MsgBox "firstname is Bill"
End If
```

That's a lot tidier. Note that we start a new line after the word Then.

1. The first line contains our condition: "If the following condition is met".
2. The second line is what we want to do if the condition is indeed met.
3. The third line tells Visual Basic that the If statement ends right here.

Try this. Delete the two quotation marks around the word Bill in the If Statement. Your code should now be this:

```
Dim firstname as String

firstname = "Bill"

If firstname = Bill Then
MsgBox "firstname is Bill"
End If
```

VB.NET puts a blue wiggly line under **Bill**. If you try to start your programme, you'll get a message box telling you that there were Build Errors, and asking if you want to continue.

Say No to return to the design environment. The reason for the blue wiggly line is that VB insists on you using double quotes to surround you text. No double quotes and VB insists it's not a string of text.

Change you code to this.

```
firstname = "Phil"

If firstname = "Bill" Then
MsgBox "firstname is Bill"
Else
MsgBox "firstname is not Bill"
End If
```

Now run the programme and see what happens when you click the button.

You should have gotten a Message Box popping up saying "firstname is not Bill". The reason is that we included the Else word. We're now saying, "If the condition is met Then display one Message Box. If the condition is not met, display a different Message Box." Notice that the Else word is on a line of it's own.

Now, after you have tested your programme, try this. Add a textbox to your form. Then change this line in your code:

```
firstname = "Phil"
```

To this:

```
firstname = Textbox1.Text
```

What the code does is to transfer the text in the Textbox directly to the **firstname** variable. We can then test what is in the variable with an If statement.

When you've finished the code, test it out by typing the word "Bill" (with a capital B) into the textbox, and then clicking the button. Then try it with a lower case "b".

So far, we've explored only simple If statements, and we're going to leave it that way for now. But they can get quite complex, because you can have one If statement inside another, and multiple Else statements.

The code you have just wrote, however, does demonstrate how you can find out what is in a variable, and take action if the condition is either met or not met. We're now going to explore another way to do that - the Select Case statement.

Select Case Statements

VB NET - Select Case

The Select Case statement is another way to test what is inside of a variable. You can use it when you know there is only a limited number of things that could be in the variable. For example, suppose we add another choice for that cream cake. We've only said that the consequences of eating the cream cake is that the Diet will be either "Ruined" or "Not Ruined". But what if we add another option - "Diet Not Tested". In other words, we ate the cake but refused to climb onto the scales to weigh ourselves!

With three choices, we can still use an If ... Else statement. But let's change it to a Select Case statement. Remember: all we are doing is testing what is inside a variable, in this case a variable called **creamcake**. Once we decide what is inside the variable, we can take some action. So let's look at how the Select Case works.

Create a Form with a button and a Textbox on it (If you have your form open from the previous section, then you can use this one). Double click the new button. You should see something like this appear.

```
Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) _
Handles Button1.Click

End Sub
```

Between the button Sub and End Sub code add the folowing

```
Dim creamcake As String
Dim DietState As String
```

```
creamcake = TextBox1.Text

Select Case creamcake

Case "Eaten"
DietState = "Diet Ruined"
Case "Not Eaten"
DietState = "Diet Not Ruined"
Case Else
DietState = "Didn't check"

End Select

MsgBox DietState
```

Run your code to test it out. Click inside your textbox and enter the word "Eaten". Then click your button to see what happens. Now enter the words "Not Eaten" adn click your button. next, try the word "eaten", with a lowercase "e".

So, how does the Select case work?

Int he code above, we first set up two variables called creamcake and DietState. Next, we transfer whatever is in Textbox1 to the variable creamcake. The Select Case begins on the next line:

```
Select Case creamcake
```

We tell Visual Basic that we want to start a Select Case statement by simply using the words "Select Case". This is enough to set up the statement. The variable **creamcake** follows the words Select Case. We're saying, "Set up a Select Case statement to test what is inside the variable called creamcake". The next line is this:

```
Case "Eaten"
```

We ask Visual Basic to check if the variable creamcake contains the word "Eaten". (Is it the Case that ... ?)

If it is the Case that creamcake contains the word "Eaten", then VB will drop down to the line or lines of code below and read that. If the variable creamcake doesn't contain the word "Eaten", the programme will skip the line or lines of code below and jump to the next Case.

The programme will continue to check all the words after Case to see if one of them contains what is in the variable creamcake. If it finds one, it will read the code below the Case word; if it doesn't find any matches, it will not do anything at all. In our code, we're checking these three Cases:

```
Case "Eaten"
Case "Not Eaten"
Case Else
```

Note also that it will only look for an exact match - "Eaten", but not "eaten".

The next line to examine is this:

Case Else

You can use the **Else** word after Case. If the programme hasn't found any matches, it will then execute the code below the Case Else line.

The final line to examing is this:

End Select

All we're doing here is to tell Visual basic to end the Select Case statement.

No more reading these lessons online - get the eBook here!

So the Select Case checks a variable for any number of different choices. If a match is found, it will then execute code below the Case option it has found. In our code, we've just told the programme to store some text in the DietState variable. After the Select Case statement has ended we displayed the variable in a Message Box.

You can use Select Case statement with numbers, as well, and the **To** word comes in very handy here. If you were checking a variable to see if the number that was in the variable fell within a certain age-range, you could use something like this:

```
Select Case agerange
Case 16 To 21
MsgBox "Still Young"
Case 50 To 64
MsgBox "Start Lying"
End Select
```

Here, a variable called agerange is being tested. It's being checked to see if it falls between certain values. If it does, then a message box is displayed.

Exercise

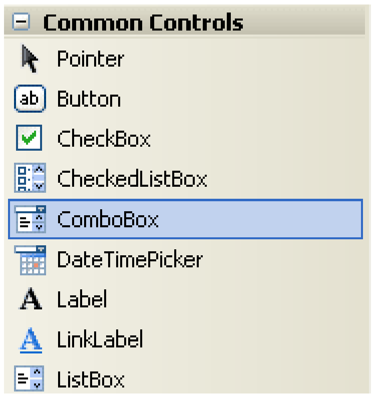
Add a new button to your form. Write a programme that tests if a person is a) A teenager, b) in their twenties, c) in their thirties, or d) none of the above.

A popular way to use Select Case statements is with a drop down box. You can then test which item a user selected from a list of available items. You're going to write a little programme to do just this. But before you can do so, you'll need to know how to add a Combo Box to a form, and how to get at the values in its list. We'll do that in the next section.

Add a Combo Box to a VB .NET form

VB NET - Combo Box

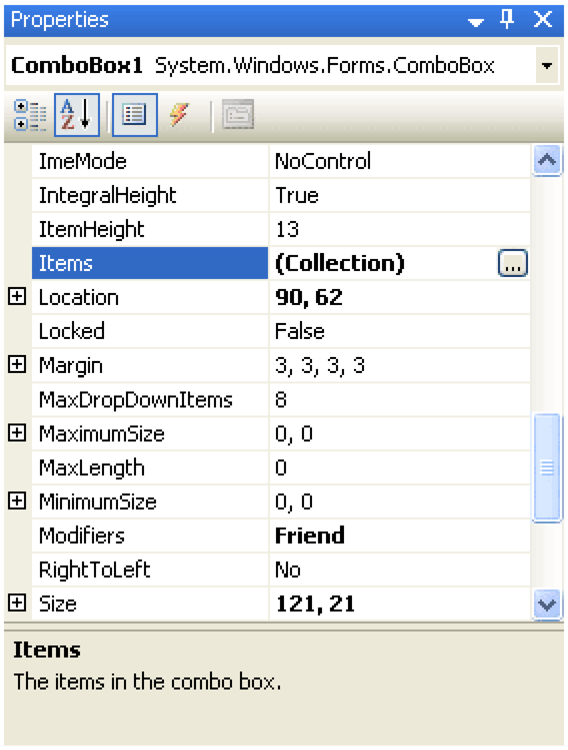
Create a new project for this section. Add a button to your new form. Then, locate the Combo Box on the Visual Basic .NET toolbar. It looks like this:



Double click the icon to add a Combo Box to your form. Or click once with the left hand mouse button, and then draw one on the form.

A combo box is a way to limit the choices your user will have. When a black down-pointing arrow is clicked, a drop down list of items appears. The user can then select one of these options. So let's set it up to do that.

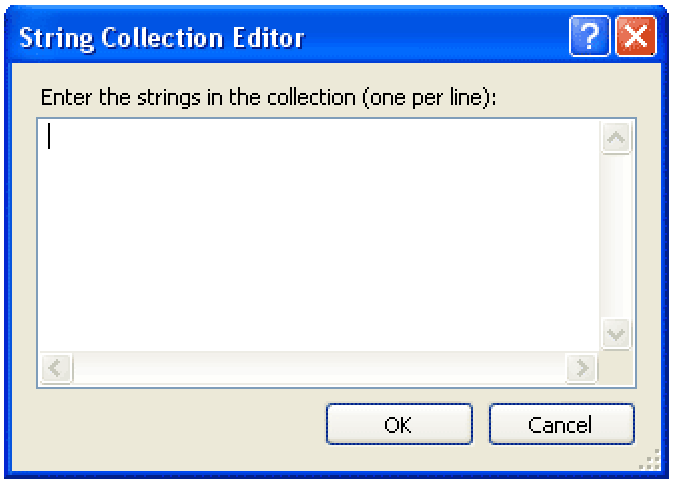
- Click on your Combo Box to select it. Then locate the Item property from the Properties Box:



ImeMode	NoControl
IntegralHeight	True
ItemHeight	13
Items	(Collection)
⊞ Location	90, 62
Locked	False
⊞ Margin	3, 3, 3, 3
MaxDropDownItems	8
⊞ MaximumSize	0, 0
MaxLength	0
⊞ MinimumSize	0, 0
Modifiers	Friend
RightToLeft	No
⊞ Size	121, 21

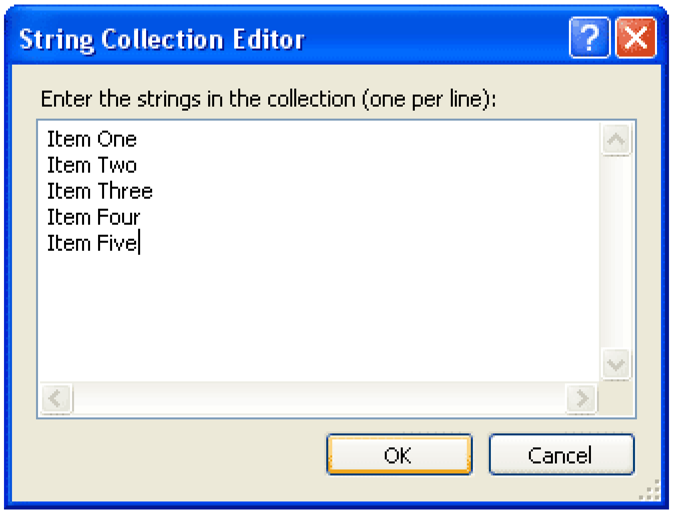
Items
The items in the combo box.

- Click the grey button, as above. The one with the three dots in it. When you do, you'll get the following box popping up:



Enter the strings in the collection (one per line):

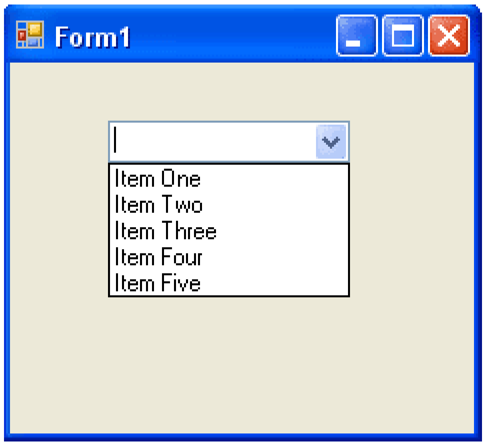
- To use the String Collection Editor, type an item and press Return (it's just like a normal textbox. Each item will be one item in your drop-down box.)
- Enter five items, as in the image below:



```
Item One
Item Two
Item Three
Item Four
Item Five
```

- Then click the OK button at the bottom.

The Editor will close, and it will look like nothing has happened. However, run your programme and test out your new Combo Box. You should have something like this:



You now need to know how to get values from the list. Once you know how to get a value from the list, you can put the value into a variable and test it with some Conditional logic.

No more reading these lessons online - get the eBook here!

Getting a value from a Combo Box is fairly straightforward, because it acts just like a Textbox. A Textbox has a Text property, and so does a Combo Box. To get a value from a Textbox, you would code like this

MyVariable = Textbox1.Text

Whatever is in the Textbox will be transferred to the variable called MyVariable. The process is exactly the same for a Combo Box. The code is like this:

MyVariable = Combobox1.Text

Now we are transferring whatever is selected from the Combo Box to the variable called MyVariable.

Let's try it. Double click the button you added to your form. This will open the code window. Then enter the following code for the button:

```
Dim MyVariable as String

MyVariable = Combobox1.Text

MsgBox MyVariable
```

Run your programme. When the programme is running, select an item from your Combo Box. Then click your button. Whatever was in the Combo Box window should have ended up in the Message Box.

And that's all there is to getting a value from a Combo Box - just access its Text Property and pass it to a variable.

Finally, the Combo Box has a DropDownStyle property. Locate this property and you'll notice its value has a drop down box. The box contains three different Combo Box styles to choose from. Experiment with all three and see how they differ.

In the next section, we'll take a look at Conditional Operators, what they are, and how to use them.

Conditional Operators

VB NET - Operators

The Conditional Operators allow you to refine what you are testing for. Instead of saying "If X is equal to Y", you can specify whether it's greater than, less than, and a whole lot more. Examine the list of Operators:

Operator Meaning **>**

This symbol means **Is Greater Than** and is used like this:

```
If number > 10 Then
MsgBox "The Number was Greater Than 10"
End If
```

<

This symbol means **Is Less Than** and is used like this:

```
If number <10 Then
MsgBox "The Number was Less Than 10"
End If
```

>=

These symbols mean **Is Greater Than or Equal to**, and are used like this:

```
If number >= 10 Then
MsgBox "The Number was 10 or Greater"
End If
```

<=

These symbols mean **Is Less Than or Equal to**, and are used like this:

```
If number <= 10 Then
MsgBox "The Number was 10 or Less"
End If
```

And

You can combine the logical operators with the word **And**. Like this:

```
If number > 5 And number < 15 Then
MsgBox "Greater than 5 And Less than 15"
End If
```

Or

You can combine the logical operators with the word **Or**. Like this:

```
If number > 5 Or number < 15 Then
MsgBox "Greater than 5 Or Less than 15"
End If
```

<>

These symbols mean **Is Not Equal to**, and are used like this:

```
If number1 <> number2 Then
MsgBox "number1 is not equal to number2"
End If
```

A word about **And** and **Or**. Notice the format with **And** and **Or**. The variable is repeated twice

```
If VariableName = 7 Or VariableName = 10 Then MsgBox "7 or 10 spotted"
```

If you just put something like this

```
If VariableName > 7 Or < 10 Then MsgBox "7 or 10 spotted"
```

then Visual Basic will give you an error message. What you have to say is

```
If [test the variable for this value] And [test the variable for that value] Then
```

Your If statement can contain an Else part, too. The format is this:

```
If [conditional logic here] Then
Some Code here
Else
```

Some Other Code here
End If

But don't worry if all that hasn't sunk in - you'll get used to the Conditional Operators as you go along. In the next part, there's two little programmes for you to write. They will test the skills you have acquired so far.

An Introduction to Loops

There are three types of loop for us to cover with VB.NET: a For loop, a Do loop, and a While … End While loop. This last one is almost the same as a Do loop, and we won't be covering it here. But the other two types of loop come in very handy, and a lot of the time you can't programme effectively without using loops.

What is a Loop?

A loop is something that goes round and round and round. If someone told you to move your finger around in a loop, you'd know what to do immediately. In programming, loops go round and round and round, too. In fact, they go round and round until you tell them to stop. You can programme without using loops. But it's an awful lot easier with them. Consider this.

You want to add up the numbers 1 to 4: 1 + 2 + 3 + 4. You could do it like this

Dim answer As Integer

answer = 1 + 2 + 3 + 4

MsgBox answer

Fairly simple, you think. And not much code, either. But what if you wanted to add up a thousand numbers? Are you really going to type them all out like that? It's an awful lot of typing. A loop would make life a lot simpler.

But don't get hung up too much on the name of the Loop. Just remember what they do: go round and round until you tell them to stop.

We'll discuss the For Loop first.

For Loops in VB .NET

This first type of loop we'll look at is called a For Loop. It is the most common type of loop you'll use when you programme. We'll use one to add up our 4 numbers, and then discuss the code. Study the following. In fact, create a new Project. Add a button to your new Form. Double click your new button and type the following code for it:

```
Dim answer As Integer
Dim startNumber As Integer

answer = 0

For startNumber = 1 To 4

answer = answer + startNumber

Next startNumber

MsgBox answer
```

Run the programme, and see what happens when you click the button. The number 10 should have been displayed in your message box.

The For loop code

We start by setting up two integer variables. We set one of these to zero. Then we start our loop code. Let's examine that in more detail.

```
For startNumber = 1 To 4

answer = answer + startNumber

Next startNumber
```

We start our loop by telling Visual Basic what type of loop we want to use. In this case it is a **For** loop:

For startNumber = 1 To 4

The next thing you have to do is tell Visual Basic what number you want the loop to start at:

For **startNumber = 1** To 4

Here we are saying "Start the loop at the number 1". The variable startNumber can be called anything you like. A popular name to call a start loop variable is the letter i (i = 1). So what we're doing is setting up a variable - the start of the loop variable - and putting 1 into it;

Next, you have to Tell Visual Basic what number to end the loop on:

For startNumber = 1 **To 4**

The **To** word, followed by a number or variable, tells Visual Basic how many times you want the loop to go round and round. We're telling Visual Basic to loop until the startNumber variable equals 4

The command that tells Visual basic to grab the next number in the sequence is this:

Next startNumber

When Visual Basic reaches this line, it checks to see what is in the variable startNumber. It then adds one to it. In other words, "Get me the next number after the one I've just used."

The next thing that happens is that Visual Basic will return to the word **For**. It returns because it's in a loop. It needs to know if it can stop looping. To check to see if it can stop looping, it skips the startNumber = 1 part, and then jumps to your end number. In our case, the end number was 4. Because **Next startNumber** adds one to whatever is in startNumber, then startNumber is now 2 (It was 1 at the start. The next number after one is ... ?).

So if startNumber is now 2, can Visual Basic stop looping? No it can't. Because we've told it to loop until it reaches number 4. It's only reached number 2, so off it goes on another trip around the loop. When the startNumber is greater than the end number, Visual Basic drops out of the loop and continues on it's way.

But remember why we're looping: so that we can execute some code over and over again.

To clarify things, change the above code to this:

```
Dim startNumber As Integer

For startNumber = 1 To 4

MsgBox("Start Number = " & startNumber)

Next startNumber
```

Run the programme, and click your button. What happens? You should have seen this in the message box, one after the other:

```
Start Number = 1
Start Number = 2
Start Number = 3
Start Number =4
```

Each time round the loop, the code for the message box was executed. You had to click OK four times - startNumber = 1 To 4.

Summary

So to sum up:

1. A For loop needs a start position and an end position, and all on the same line
2. A For loop also needs a way to get the next number in the loop
3. A loop without any code to execute looks like this:

```
For i = startNumber To endNumber

Next i
```

The above code uses two variables for the start and end numbers. The start number for the loop goes directly into the variable called i. When Visual Basic wants the next number, it just add one to whatever is in the variable i. You could use it like this:

```
Dim startNumber As Integer
Dim endNumber As Integer
Dim i As Integer

startNumber = 1
endNumber = 4

For i = startNumber To endNumber
Msgbox i
Next i
```

Change the code for your button to that new code, and test it out. Study the code so that you understand what is going on.

For Loops might not be easier to understand than just typing **answer = 1 + 2 + 3 + 4**, but they are a lot more powerful if you want to add up a thousand numbers!

Exercise

Put two textboxes on your form. The first box asks users to enter a start position for a For Loop; the second textbox asks user to enter an end position for the For loop. When a button is clicked, the programme will add up the numbers between the start position and the end position. Display the answer in a message box. You can use this For Loop code

```
For i = startNumber To endNumber

answer = answer + i

Next i
```

Get the startNumber and endNumber from the textboxes.

Exercise

Amend your code to check that the user has entered numbers in the textboxes. You will need an If statement to do this. If there's nothing in the textboxes, you can halt the programme with this code

```
Exit Sub
```

For this exercise, you will be passing whatever is in the textboxes to integer variables. It is these variables you are checking with your If Statement. Because numbers will be entered into the textboxes, remember to convert the text to a value with Val( ).

But the Text property will return a zero if the box is empty. So your If statement will need to check the variables for a value of zero. If it finds a zero, Then you can use the Exit Sub code. The If statement should come first, before the For Loop code.

Do Loops in VB .NET

We saw with the For Loop that a specific number of times to loop was coded into it. We said

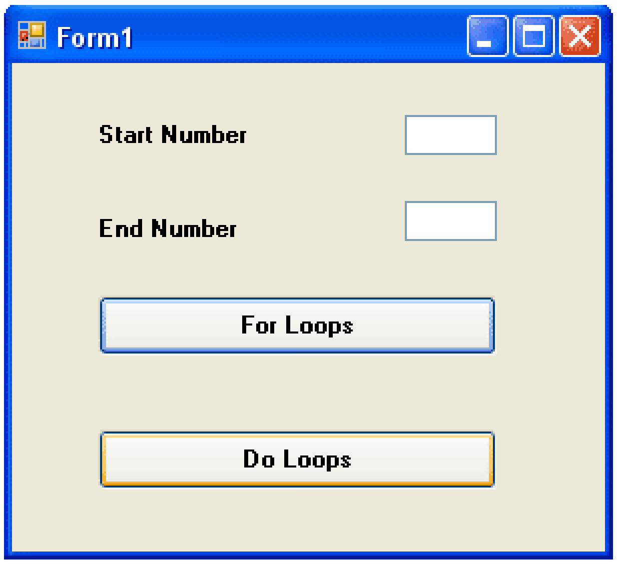
```
For startNumber = 1 To 4
```

We knew that we only wanted to loop four times. But what if we don't know how many times around the loop we want to go? Later, we'll be opening text files and reading the data from them. If we used a For loop to get every line of text, we'd have to know before hand how many lines the text file held. A For Loop would not be very efficient in this case.

But a Do Loop would be. With a Do Loop we can use word s like "While" and "Until". And then we can say, "Go round and round the loop While there's still text to be read from the file." An example might make things clearer.

Load the form you created for the last exercise, the one that has two textboxes and a Button and tested your understanding of For loops.

Add another button to the Form. Your form might look something like this:



Double click the new button to open the code window, and then type the following code for the new button:

```
Dim number as Integer

number = 1

Do While number < 5
MsgBox number
number = number + 1
Loop
```

When you've finished, run the programme and see what happens. The numbers 1 to 4 should have displayed in your message box.

So the Do loop keeps going round and around. And when it gets to the top, it tests the "While" part - **Do While number is Less Than 5**. It's really answering a Yes or No question: is the number inside the variable called **number** Less Than 5? If it is Less Than 5, go round the loop again. If it's not Less than 5, Visual Basic jumps out of the Loop entirely.

You can add the "While ... " part to the bottom, just after the word "Loop". Like this:

```
Do

number = number + 1

Loop While number < 5
```

Try it and see what difference it makes.

None, right? But there is a difference between the two. With the "While ... " part at the bottom, the code above it gets executed at least once. With the code on the first line after the word "Do", the code might not get executed at all. After all, the number inside the variable might already be Greater Than 5. If it is, Visual Basic won't execute the code.

Do ... Until

You have another choice for Do Loops - Do ... Until.

There's not much difference between the two, but a Do ... Until works like this. Change your Loop code to the following:

```
Do Until number < 5

MsgBox number
number = number + 1

Loop
```

Run the code and see what happens.

Nothing happened, right? That's because we "Keep looping UNTIL the number in the variable called number is Less Than 5" The problem is, the number inside the variable is already Less Than 5. And if the number is Less than 5, then the code won't execute - because it has already met the end condition.

Change that Less Than sign to a Greater Than sign, and then test your code again. Now what happens?

The numbers 1 to 5 should have displayed. Again, the loop keeps going round and around testing to see if our end condition is met, in this case Is Greater Than 5. If the condition is met, VB breaks out of the Loop; if not, keep going round.

Change the Greater Than sign to Greater Than or Equal to (>=), and test it again. It should now print 1 to 4.

The "Until" part can go at the bottom, just after the word Loop. Like this

```
Do

MsgBox number
number = number + 1

Loop Until number >= 5
```

To sum up, use a Do Loop if you don't know what the end number is going to be, otherwise a For Loop might be better.

You're now going to write a programme that uses a For Loop inside a Do Loop. The programme works out the times table.

A Times Table Programme

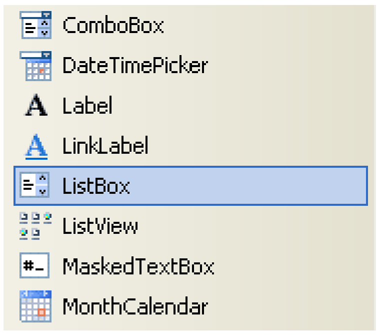
Start a new project for this. Onto your new Form, place two textboxes and a Button. Set the Text property of Textbox1 to 1, and the Text property of Textbox2 to 10. Set the Text property of the Button to "Go".

When the Go button is clicked, the programme will put the numbers from the Textbox into two variables. We'll then put a value into a variable called multiplier. If you're doing the times tables, the format is

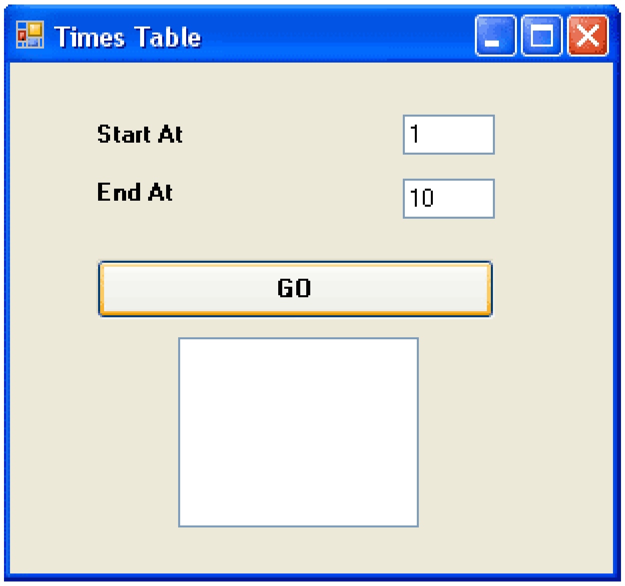
X multiplied by Y = Z
(2 multiplied by 3 = 6)

We'll use a Do Loop to work out the multiplier (that's the Y part); a For Loop will work out the rest. We'll then display the results in something called a Listbox.

So add a List Box to your form. It looks like this in the toolbox:



The form you design should look something like this one:



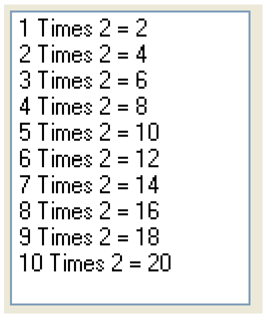
A List box is similar to a Combo Box, in that you have a list of items that the user can select. Here, we're just using it display the results of our programme. We'll add items to the List box with our code, rather than in design time like we did for the Combo box.

So, here's the code for the entire programme. Double click your Go button, and add the following:

```
Dim number1 As Integer
Dim number2 As Integer
Dim multiplier As Integer
Dim answer As Integer
Dim i As Integer

number1 = Val(TextBox1.Text)
number2 = Val(TextBox2.Text)

multiplier = 2

Do While multiplier < 3

For i = number1 To number2

answer = i * multiplier
ListBox1.Items.Add(i & " Times " & multiplier & " = " & answer)
Next i

multiplier = multiplier + 1
```

Loop

When you've finished, run the programme and see how it works. You should see this appear in your List box:



```
1 Times 2 = 2
2 Times 2 = 4
3 Times 2 = 6
4 Times 2 = 8
5 Times 2 = 10
6 Times 2 = 12
7 Times 2 = 14
8 Times 2 = 16
9 Times 2 = 18
10 Times 2 = 20
```

Let's run through the code to see how it works.We'll do that on the next page. Click the link below to move on.

The Times Table Code

As you can see from our last lesson, we've set up five Integer variables - number1, number2, multiplier, answer and i.

The next thing we did was to pass whatever is in our two Textboxes straight into the two variable, number1 and number2. The start number goes into textbox one, and the end number goes into textbox2.

number1 = Val(TextBox1.Text)
number2 = Val(TextBox2.Text)

In the next line of code, we set a starting value for the multiplier variable:

multiplier = 2

Then we had our two Loops, one inside the other. The first is the Do Loop:

Do While multiplier < 3

multiplier = multiplier + 1

Loop

This Do loop is exactly the same as the one you met before. All is does is go round and round While the variable called **multiplier** is Less Than 3. The bit of code between Do and Loop just keeps adding one to whatever is in **multiplier** (incrementing the variable).

The other thing the Do Loop does is to execute our second loop - the For Loop. The For Loop gets executed each time around the Do Loop. The For Loop was this:

```
For i = number1 To number2

answer = i * multiplier
ListBox1.Items.Add(i & " Times " & multiplier & " = " & answer)

Next i
```

Remember: the **number1** and **number2** variables hold our numbers from the Textboxes. We set these to 1 and 10. So our first line of the For Loop is really this:

```
For i = 1 To 10
```

We're saying, "Start a For Loop". Whatever is in the variable called number1, make that the starting number for the Loop. Put this value into the variable called i. The end of the Loop will come when the variable called i has the value 10. Stop looping when you reach this value.

The next part of the code reads this:

```
answer = i * multiplier
```

This means, Put into the variable called answer the following sum: whatever is in the variable called i multiplied by whatever is in the variable called multiplier.

(For a longer discussion on how to use the basic Maths symbols in VB .NET, see here: basic Maths symbols in VB .NET)

So that says, put into the variable answer the sum of i times multiplier. We're getting whatever is in i from the For loop. But we're getting whatever is in the variable called multiplier from the Do Loop.

So Visual Basic will read the Do While line after noting that the number 2 has been put into the variable called multiplier. It will then check to see if the end of the loop condition has been met - multiplier is Less Than 3. As 2 is not Less Than 2, VB drops down to the next line.

The next line is a For Loop - For i = 1 To 10. This entire loop will be executed, and the code inside the For loop worked out.

So the For loop gets executed 10 times. Which means that our sum will get executed 10 times. When the end condition of the For loop is met, Visual Basic will exit the For Loop and drop down to the line of code below Next i. The code below Next i is this:

```
multiplier = multiplier + 1
```

What was inside the variable called multiplier was the number 2, so 2 + 1 = 3. And three is the new number inside the variable multiplier.

Then, Visual Basic drops down to the next line, which is Loop. So it goes back up to the Do While line of code

Do While multiplier < 3

The same question is asked again: Have we met the end condition for the Do Loop? The end condition was "Keep looping while the variable called multiplier is Less Than 3".

Because the value in multiplier is now equal to 3 and not less than 3, the end condition has been met. When the end condition is met, VB exists the Do loop.

No more reading these lessons online - get the eBook here!

Exercise

Change the Do While line to this:

Do While multiplier < 5

Can you guess what will appear in your list box? Run your programme and find out.

Finally, a word about the line that displays your text in the list box. It was this:

ListBox1.Items.Add(i & " Times " & multiplier & " = " & answer)

To add items to a list box with code, first you type the name of your list box:

ListBox1

Type a full stop and a drop down list will appear. Select Items from the list.

ListBox1.Items

Type another full stop and again a drop down list will appear. Select the Add Method

ListBox1.Items.Add

This method, not surprisingly, lets you add items to your list box. Whatever you want to add goes between a pair of round brackets:

ListBox1.Items.Add( )

In between the round brackets, we have this for our code:

i & " Times " & multiplier & " = " & answer

It might be a bit long, but there are 5 parts to it, all joined together by the concatenate symbol (&):

i
" Times "
multiplier
" = "
answer

The variable i holds the current value of the For Loop; " Times " is just direct text; multiplier holds the value we're multiplying by (our times table); " = " is again direct text; and answer is the answer to our times table sum.

If you want to clear the items from a List box you can do this. At the top of the code, enter this line:

ListBox1.Items.Clear()

So instead of selecting Add from the final drop down list, select Clear.

Exercise

Add another textbox to your form. This will represent the "times table". So far you have been getting this value directly from the code. For this exercise, get the multiplier value directly from the textbox. Add appropriate labels to all your textboxes.

The multiplier variable is the starting point for your "times tables". What else do you need to amend in the code? Think about what happens if you entered a 5 into your new Textbox.

If you enter a number 5 or greater, the code doesn't get executed! Can you understand why that is? If not, I suggest you go over this looping section again.

Exercise

Amend the above code to solve the problem of the programme not being executed.

If you successfully solved the problem, what else did you notice? Assuming that you added two labels to your new textboxes - "Start At" and "End At" - what happens if you enter a 3 in the

Start At textbox and a 5 in the End At textbox? Will you actually get the 5 times table? What can you do so that the 5 times table is displayed? (Consider using a different Do Loop. What about a Do ... Until loop?)

In the next part, we'll take a closer look at the basic Maths symbols you can use in VB .NET.

The Basic Math Symbols in VB .NET

If you're doing any programming, you need to know how to use the basic Math symbols. The basic Math symbols in Visual Basic .NET are these:

+ The Plus sign adds numbers together **-** The minus sign takes one number away from another ***** The symbol above the number 8 on your keyboard tells Visual Basic to multiply two numbers **/** The forward slash on your keyboard is the divide by symbol **=** The equals sign
A word or two about how to use the mathematical symbols in Visual Basic. You can use the operators by themselves:

```
answer = 8 + 4
answer = 8 - 4
answer = 8 * 4
answer = 8 / 4
```

Or you can combine them by using round brackets (parentheses).

Here, Visual Basic will work out the sums in parentheses first, and then add the two sums together

```
answer = (8 - 4) + (4 -2)
answer = 4 + 2
answer = 6
```

But you've got to be careful with parentheses, because there is a strict order that VB uses when doing its Math. Consider this sum

answer = 8 - 4 + 4 + 2 * 2

Try that code behind a new button. Display the result in a MsgBox. What answer did you get? Was it the answer you were expecting?

If you didn't pick up much Math in school, you might do the sum from left to right, thereby giving you an answer of 20.

```
8 - 4 = 4
+ 4 = 8
```

+ 2 = 10
* 2 = 20

But VB doesn't work it out like that. Visual Basic will do the multiplying first. So it will calculate like this:

2 * 2 = 4
8 - 4 + 4 = 8
8 + 4 = 12

But try changing the code to this:

```
answer = (8 - 4) + (4 + 2) * 2
```

Here, we've added some parentheses.

Run the code and see what happens. That's right - you get 16! This time, Visual Basic will do the (4 + 2) * 2 part first, and then add that to 8 - 4. Which gives you 16.

Try this code and see what happens:

```
answer= ((8 - 4) + (4 + 2)) * 2
```

Now we get an entirely different answer - 20! The parentheses above have grouped our sums into separate sections, and we now get a third solution to our seemingly simple calculation.

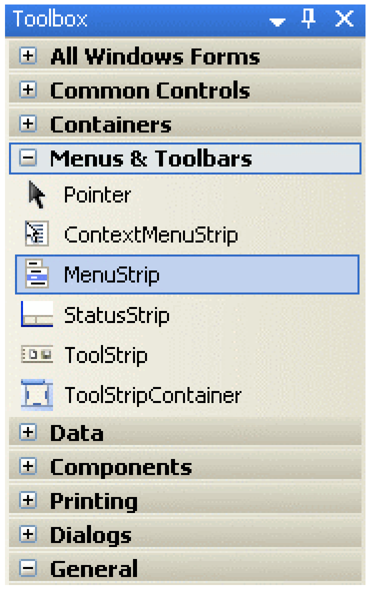
The lesson to learn here is to take care when mixing different Math symbols. Use parentheses to spell out to VB exactly what you mean.

In the next section of the course, we'll have some fun adding menus to a Visual basic .NET form.

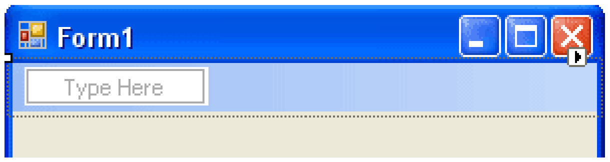
Adding Menus to a Visual Basic .NET Form

In this section we’ll see how to add menus to a Visual Basic .NET form. These type of menus are very common to Windows programme. Visual Basic itself has many of these drop down menus - File, Edit, View, Project, Format, etc. And they’re very easy to add.

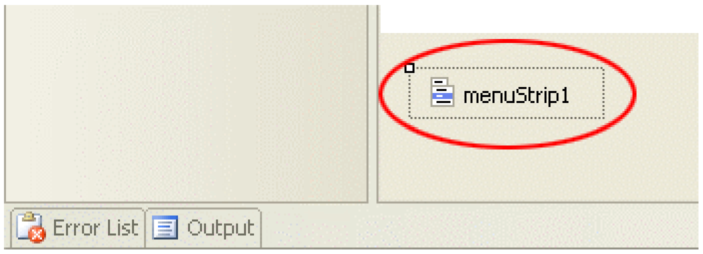
Start a new project. To your new form, use the toolbox to add a MenuStrip control:



Double click the control to add one to your form. When you do, you'll notice two things. At the top of your form, you'll see this:

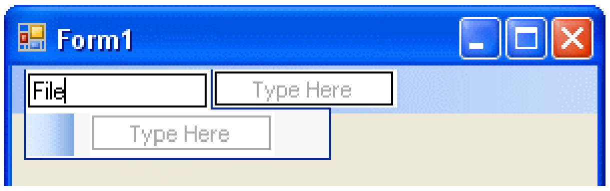


We'll see how to construct our menu soon. But notice the other things that gets added to your project. Examine the bottom of your screen, on the left. You'll see this:

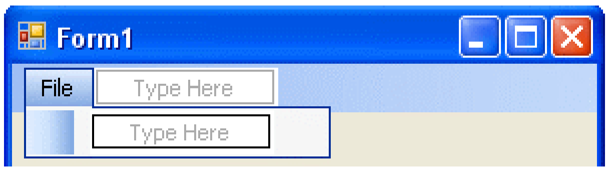


This is the control itself. If you click on this (it's highlighted above), you'll see that the Properties box on the right changes. There are many properties for the control. But there are lots of properties for the MenuItem object. The MenuItem object is the one at the top of the form - The one that says Type Here.

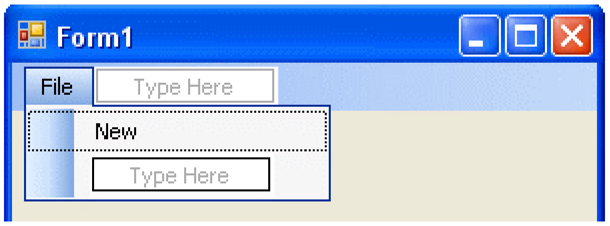
To start building your menu, click inside the area that says "Type Here". Type the word File:



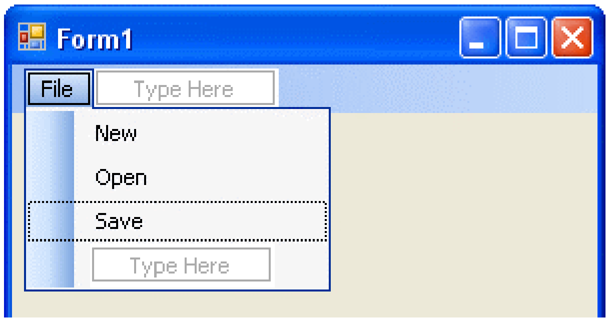
Now press the enter key on your keyboard. Your menu will look like this:



To create items on your File menu, click inside the **Type Here** box. Enter the word **New**, and press the enter key on your keyboard again. Your menu will then look like this:

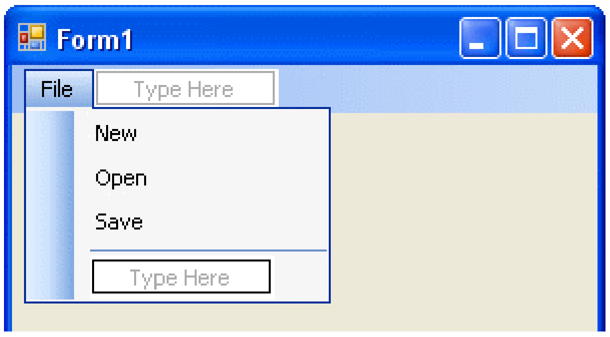


Add an "Open" and a "Save" item to your menu in the same way. It should look like this:

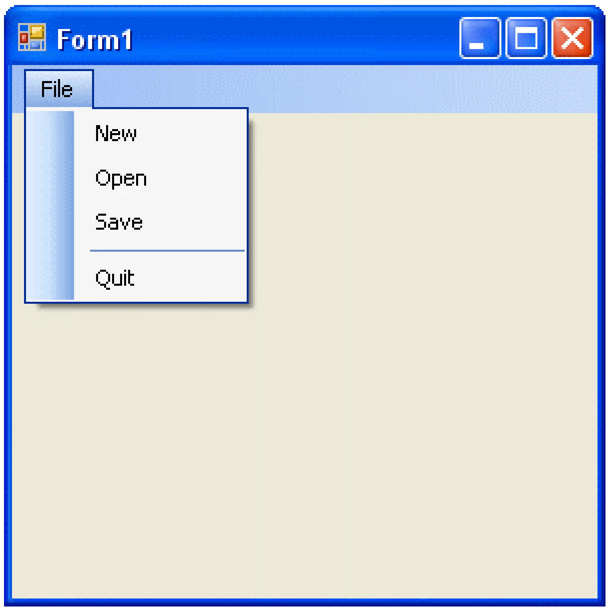


The final item we'll add to our menu is an "Exit" item. But you can add a separator between the "Save" and "Exit".

To add a separator, click inside the blue "Type Here" box. Instead of typing a letter, type the minus character "-" (in between the "0" key and the "+/=" key on your keyboard). When you hit your return key, you'll see the separator appear:



Click inside the "Type Here" area, and add an Exit (or Quit) item. Click back on your form, away from the menu, to finish off. You should now have a File menu like this one:



To see what your menu look like, Run your programme. Click the File menu. We haven't added any code to the menu yet, so nothing will happen if you click an item on the menu. But it does look quite good. Very professional!

In the next part, we'll see how to add some code to the Quit menu.

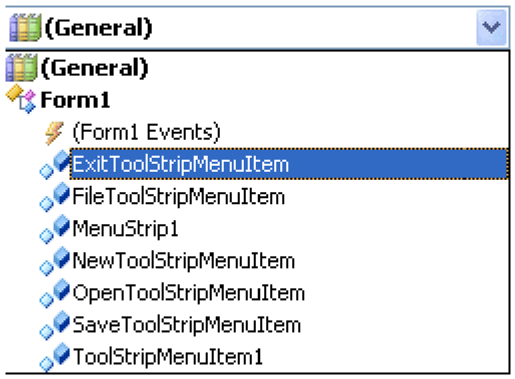
Adding code to a VB.NET menu

This tutorial assumes that you have completed the previous one.

Stop your programme and return to the design environment. Click **File** in Design Time to see your drop down menu. You can double click an item to open up the code window. But don't do that yet.

Another way to get to the code for an object is this:

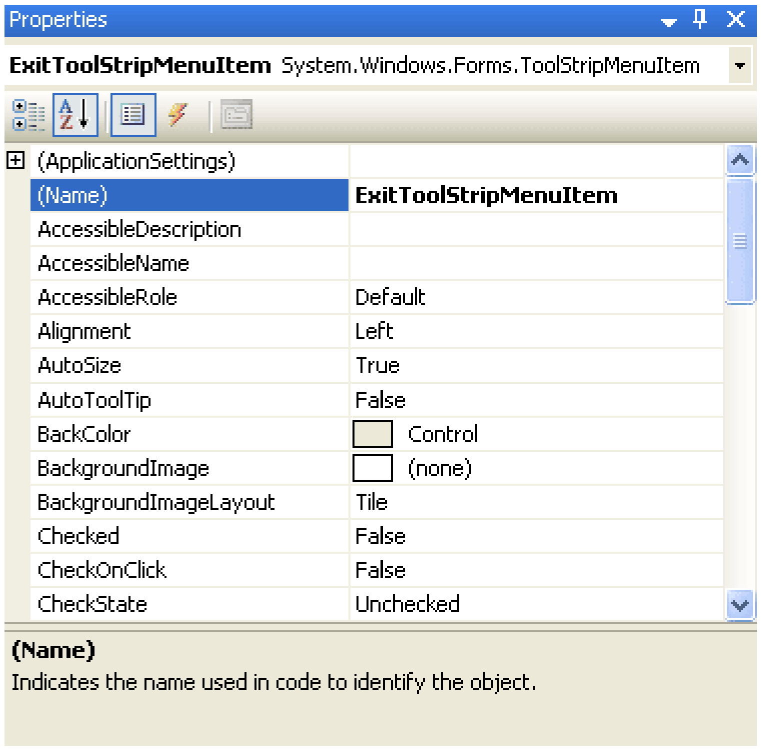
- Press F7 on your keyboard to go to the code window
- Click the black arrow at the top, where it says **General**:



The Exit menu here is "ExitToolStripMenuItem". If you were to click that item, a code stub would open, ready for you to type your code.

However, "ExitToolStripMenuItem" is very difficult to remember. We can rename our menu items so that they are more descriptive. So do this:

- Get back to your form by pressing Shift + F7 on your keyboard
- Click the **File** menu to select it
- Select your **Exit** (or your Quit) item (Careful not to click in the middle as this may open the code window.Click near the left edge somewhere.)
- When you have the Exit item selected, look at the properties box on the right:



Properties

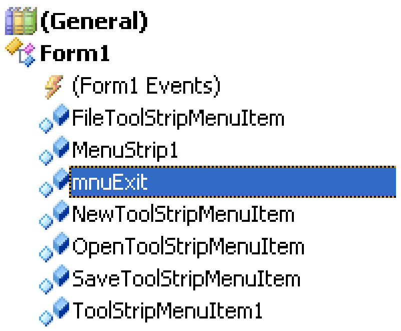
ExitToolStripMenuItem System.Windows.Forms.ToolStripMenuItem

(ApplicationSettings)	
(Name)	ExitToolStripMenuItem
AccessibleDescription	
AccessibleName	
AccessibleRole	Default
Alignment	Left
AutoSize	True
AutoToolTip	False
BackColor	Control
BackgroundImage	(none)
BackgroundImageLayout	Tile
Checked	False
CheckOnClick	False
CheckState	Unchecked

(Name)
Indicates the name used in code to identify the object.

- Click inside the **Name** property
- Change it to **mnuExit** (or mnuQuit)
- Press your return key on your keyboard to confirm the change

Now press F7 again to bring the code window up. Click the drop down arrow of the General box, and you should see the new name appear (Notice that MenuItem6 has vanished):

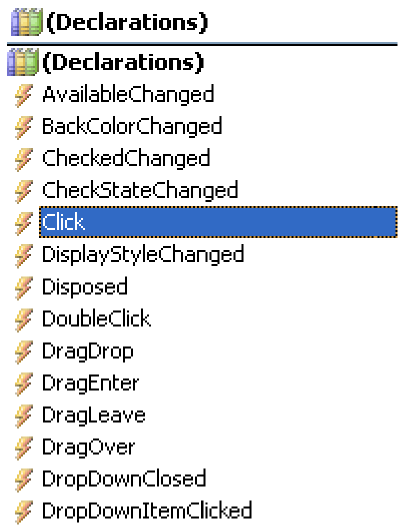


(General)
Form1
(Form1 Events)
FileToolStripMenuItem
MenuStrip1
mnuExit
NewToolStripMenuItem
OpenToolStripMenuItem
SaveToolStripMenuItem
ToolStripMenuItem1

Click on your new mnuExit item.

Nothing will happen!

To jump straight to the code, you need to look at the drop down box opposite. It will probably say "Declarations". Click the arrow and you'll see a new list:



The items in the Declarations box are called Events. The Event you want is the **Click** event. So select that one from the list (we'll cover Events in more depth later). When you select Click from the list, you are taken straight into the code for that event. It should be like this one:

```
Private Sub mnuExit_Click(ByVal sender As Object, _
                           ByVal e As System.EventArgs) _
                           Handles mnuExit.Click

End Sub
```

The code above has been tidied up to fit on this page; yours will all be on one line. But notice that it says **mnuExit_Click**.

Don't worry too much about what it all means; we'll get to that in a later section. What we want to do is add some of our own code, so that out Exit menu item actually does something.

There's only one line of code to add. It's this:

Me.Close( )

The word "Me" refers to the form. When your type the word Me, you'll see a list if items appear. Double click the word Close, then press your return key. Your code window should look like this:

```
Private Sub mnuExit_Click(ByVal sender As Object, _
                           ByVal e As System.EventArgs) _
                           Handles mnuExit.Click

    Me.Close()

End Sub
```

(Again this has been tidied up to fit on this page.)

To test out your new code, run your programme. Click your **File** menu, and then click the **Exit** item. Your form should close down, and you'll be returned to the design environment.

In the next part, we'll see how to add Sub menus.

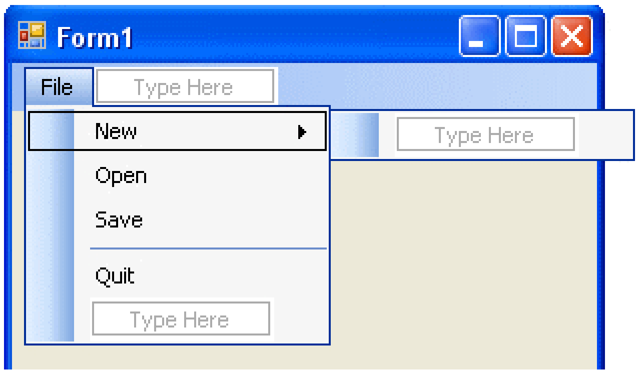
Add a Sub Menu to your VB.NET Form

This tutorial assumes that you have completed the first one.

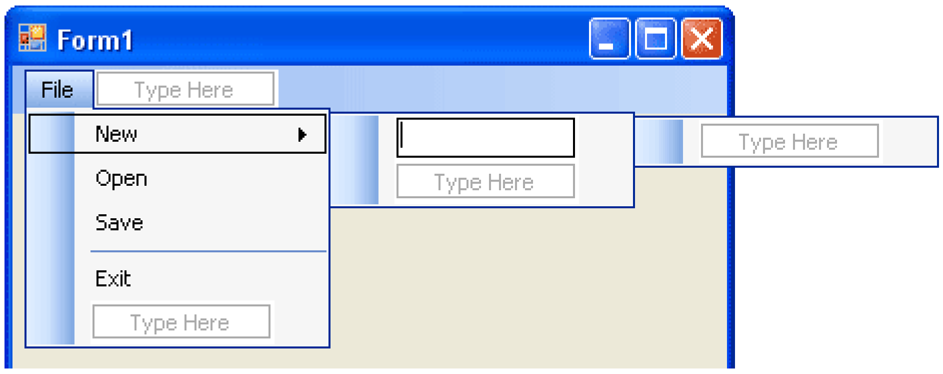
A sub menu is one that branches of a menu item. They usually have an arrow to indicate that there's an extra menu available. You will have seen these plenty of times in Windows programmes.

You can create our own sub menus quite easily. Try this:

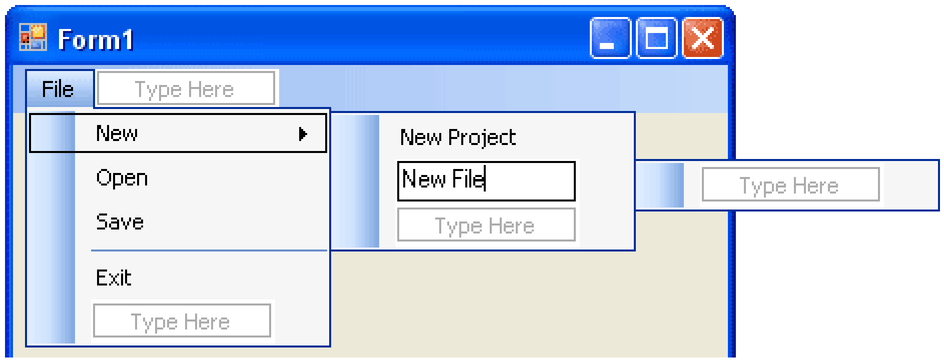
- Return to the Form view (Shift + F7 is a shortcut)
- Click on your File menu so that you can see it all
- Select the New item (Careful where you click. Click once on the left edge). You should see this:



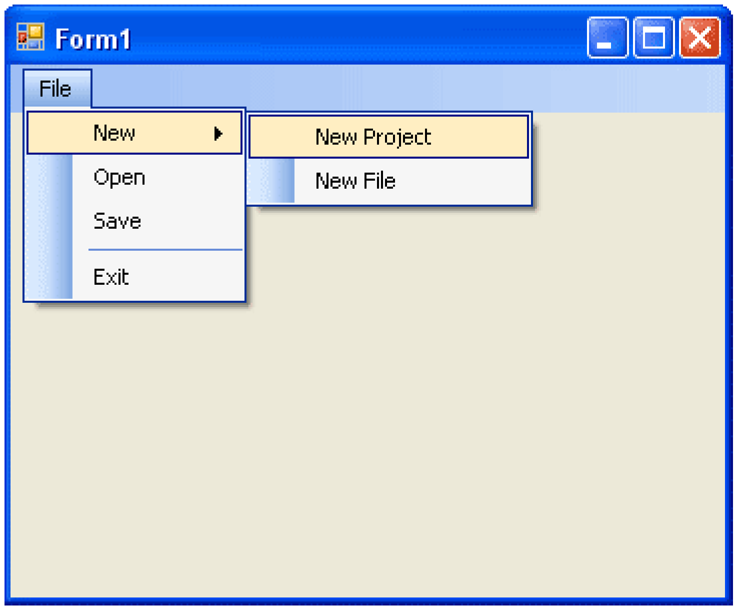
- Click on the "Type Here" just to the right of New
- You'll see yet more "Type Here" areas:



- Type **New Project**, and then hit the return key on your keyboard
- Type in **New File** and then click away from the menu, somewhere on the form
- You will then have a menu like this one:



- Save your work, and then run your programme. Click your new menu to see the following:



Of course, none of the menu items work except the Exit menu. But you should have found that adding menus to your programmes is an easy matter with VB.NET.

One more thing we can do. If you look closely at a lot of menu items, you see that they have shortcuts attached. There's two types of shortcuts: An underline shortcut, and a key combination shortcut. We'll see how to do that in the next section.

Add Shorcuts to your Menu Items

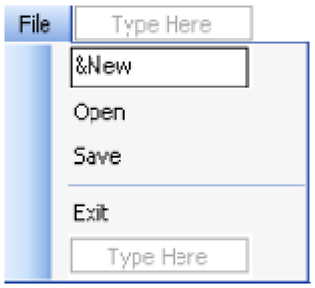
This tutorial assumes that you have completed the first one.

In the previous tutorial, you learned how to add Sub Menus to your VB .NET forms. In this part, you'll see how to add shortcuts to your menu items

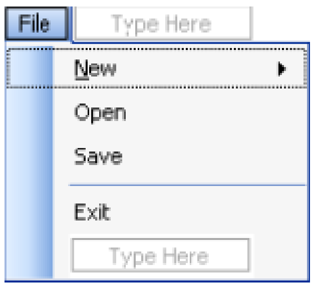
Underline Shortcut

To add an underline, do this:

- Click on your **New** menu item once. This will select it
- Position your cursor before the "N" of **New**
- Type an ampersand symbol (**&**)

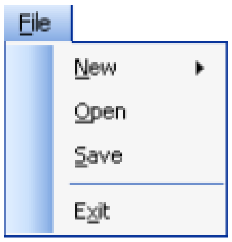


- Hit the return key on your keyboard
- You should see this:



Notice that "N" of **New** is now underlined. If you want an underline shortcut, the ampersand character should be typed before the letter you want underlined.

Add underlines for the "F" of you **File** menu, the "O" of **Open**, the "S" of **Save**, and the "X" of **Exit**. When you're done, your menu should look like this one:



Time to see if your shortcut works. Run your programme. To use the underline shortcuts on menus, you first hold down the Alt key on your keyboard. Then type the underline character.

- Hold down the Alt key while your programme is running (You might not be able to see the underlines until you press the Alt key.)
- Press the letter "F" on your keyboard
- Then press the letter "X" (for the Exit menu)
- Your programme should close down

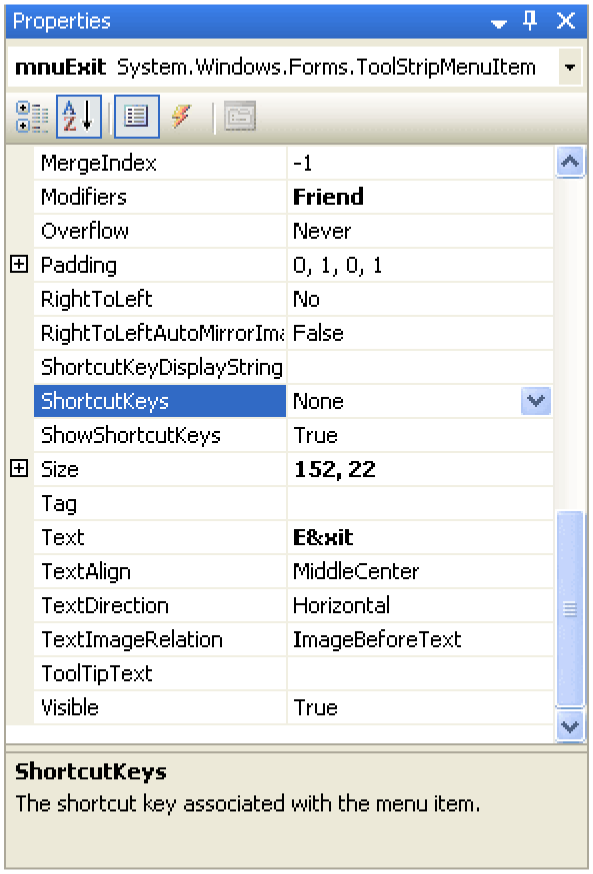
All that and you didn't have to write a single line of code!

No more reading these lessons online - get the eBook here!

Key combination shortcuts

A key combination shortcut is one that appears at the end of a menu item (Ctrl + X, for example). You can easily add this option to your own programmes. So try this:

- In Design time, select the Exit item on your menu
- Look at the properties box on the right
- Locate the **ShortcutKeys** item:



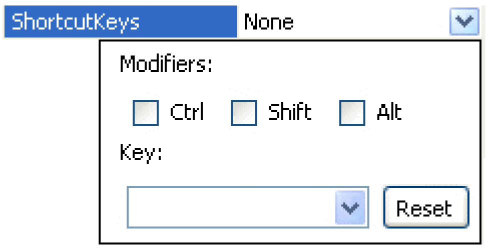
Properties

mnuExit System.Windows.Forms.ToolStripMenuItem

MergeIndex	-1
Modifiers	**Friend**
Overflow	Never
⊞ Padding	0, 1, 0, 1
RightToLeft	No
RightToLeftAutoMirrorIm	False
ShortcutKeyDisplayString	
ShortcutKeys	None
ShowShortcutKeys	True
⊞ Size	**152, 22**
Tag	
Text	**E&xit**
TextAlign	MiddleCenter
TextDirection	Horizontal
TextImageRelation	ImageBeforeText
ToolTipText	
Visible	True

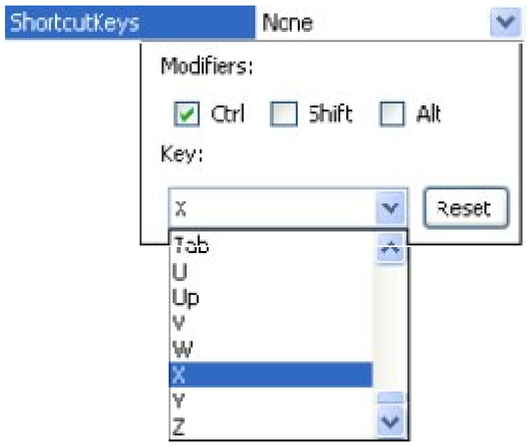
ShortcutKeys
The shortcut key associated with the menu item.

- Click the down arrow to reveal the following:

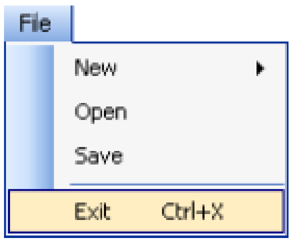


ShortcutKeys None

The Modifier is the key you press with your shortcut. For example, the CTRL key then the "X" key on your keyboard. Place a check inside the Ctrl box. Then select the letter "X" from the Key dropdown list, as in the next image:



Click back on your menu to see what it looks like:



Run your programme and test out the shortcut. Don't click the File menu. Just hold down the Ctrl key on your keyboard. Then press the letter X. Again, the programme will close down.

You can add these types of shortcuts to any menu item. Just remember to choose a different key stroke combination for each one. And don't have too many of them - they'll spoil the look of your menu!

Now that you know how to add Menus, Sub Menus and Shortcuts it's time for you to complete your own menu bar. Click the project below to see what you need to do. It's not very difficult! In the section after the project, you'll see how to add code for your new menu bar.

A VB .NET Menu Project

This tutorial assumes that you have been following along from the first one.

Add the following **Main** Menu items to the menu bar you have already designed in this section:

1. Edit
2. View

On your Edit Menu, place the following menu items:

- Undo
- Cut
- Copy
- Paste

On your View Menu, place the following menu items:

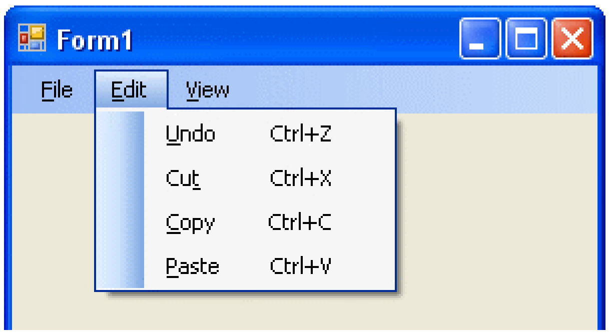
- View Textboxes
- View Labels
- View Image

Just like you did with the Exit menu item, Change the Name property of ALL menu items. Do not leave them on the defaults of "MenuItem1", "MenuItem2", etc. (You should change the Name property to something relevant, and use the prefix mnu. For example, the Undo item could have the Name mnuUndo.)

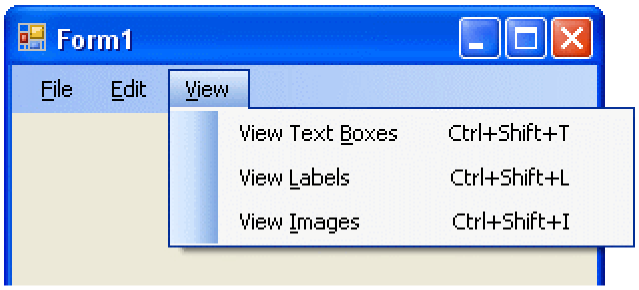
- Add an underline shortcut for ALL menu item
- Add a least one key combination shortcuts per drop down menu (you already have one on the File menu, so this doesn't count)

When you have finished, your menus should look like these (though you can use different key combinations, if you like):

Edit Menu



View Menu



- Write code to display a message box whenever a menu item is clicked, or its shortcut used. The message box should explain what the menu item will do when it's fully implemented.

There's only one line of code to write for each menu item. You can get at the code for the click event of each menu item in exactly the same way that you did for the Exit menu item.

Good Luck!

In the next part, you'll learn how to write code for all your new menu items. We start with the **Open** menu item.

The Open File Dialogue Box

For the last project, you designed a form interface that had a File, Edit and a View menu. In this section, we'll write code so that your menu items actually do something other than displaying message boxes. In other words, the **Edit > Cut** menu will really cut text, and the **Edit > Paste** menu will really paste text.

So open up the project you completed for the previous section. Comment out or delete any message box code. (You comment out code by typing a single quote character at the start of the line. The line will then turn green, and will be ignored when the programme is run.)

We'll start with the **File > Open** menu.

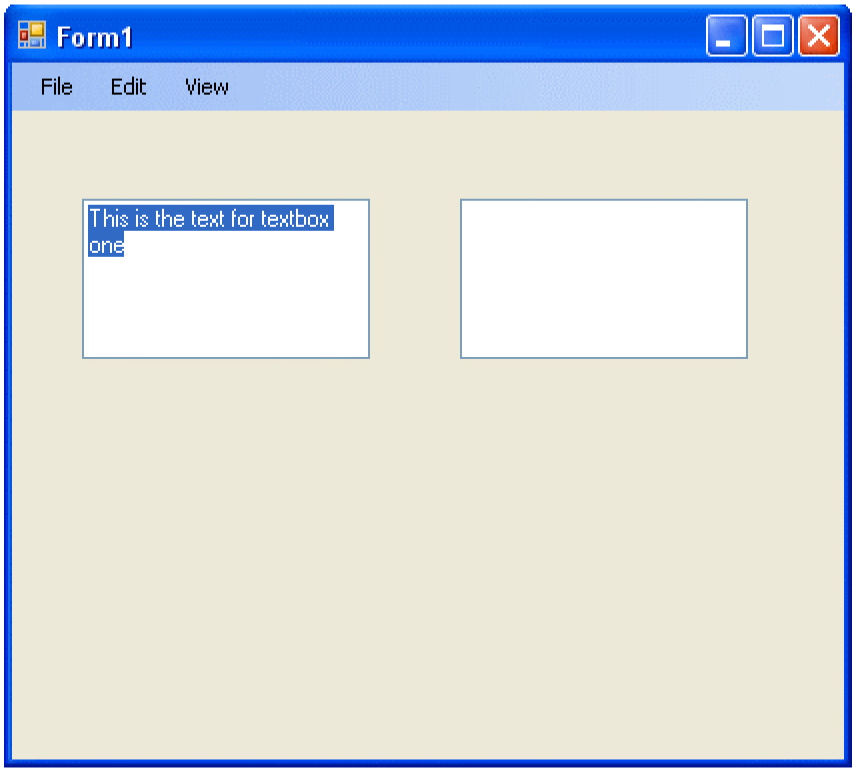
The Open File Dialogue Box

In most programmes, if you click the **File** menu, and select the **Open** item, a dialogue box is displayed. From the dialogue box, you can click on a file to select it, then click the **Open** button. The file you clicked on is then opened up. We'll see how to do that from our menu. (Except, the file won't open yet - only the dialogue box will display, and then name of the chosen file. You'll learn how to open files in a later section.)

First, place two textboxes on your form. In the properties box, locate the MultiLine property. It is set to False by default (which is why you can't change the height of textboxes). Change this value to **True**.

Type some default text for the Text Property of textbox1. Change the Font size to 14 points.

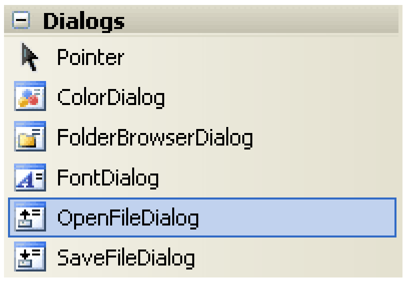
Your form should now look something like this one:



We'll work with these textboxes when we do the **Edit** menu. So let's leave them for now.

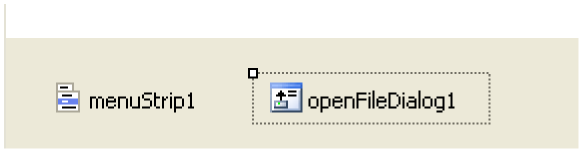
When we click on **File** > **Open** from our menu, we want the Open dialogue box to appear. This is fairly straightforward in VB.NET. In fact there is even a control for it!

Open up your toolbox, and locate the control called "**OpenFileDialog**". You might have to scroll down to see it. But you're looking for this:



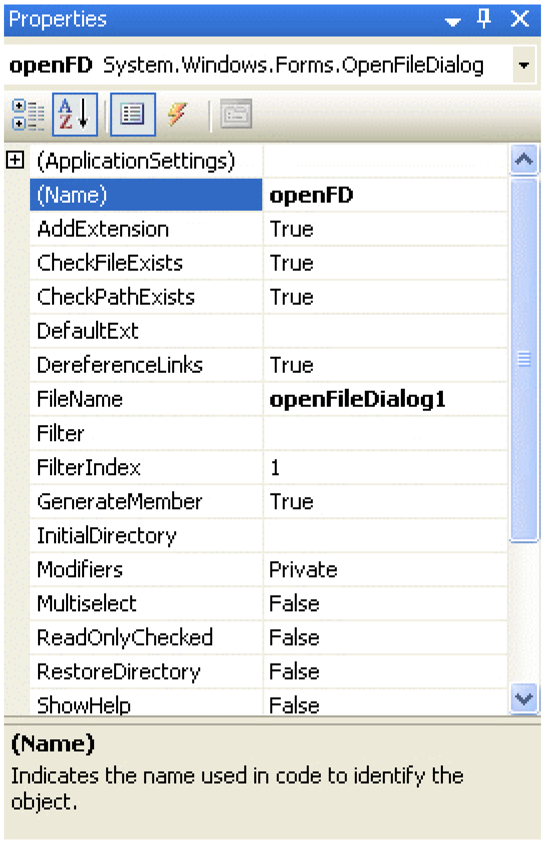
Double click the control to add one to your project.

But notice that the control doesn't get added to your form. It gets added to the area at the bottom, next to your menu control:



The shaded area surrounding the control means that it is selected. If you look on your right, you'll see the properties that you can use with the control.

Click on the **Name** property and change the name to openFD. When you change the name in the properties box, the name of the control at the bottom will change:



openFD System.Windows.Forms.OpenFileDialog

(ApplicationSettings)	
(Name)	openFD
AddExtension	True
CheckFileExists	True
CheckPathExists	True
DefaultExt	
DereferenceLinks	True
FileName	openFileDialog1
Filter	
FilterIndex	1
GenerateMember	True
InitialDirectory	
Modifiers	Private
Multiselect	False
ReadOnlyChecked	False
RestoreDirectory	False
ShowHelp	False

(Name)
Indicates the name used in code to identify the object.

We'll now write some code to manipulate the properties of our new control. So do the following:

- Access the code for your File > Open menu item. (To do this quickly, you can simply double click the Open item on your menu bar. Or, press F7 to access the Code View.)
- Click the name of your menu item from the left drop down box at the top of the code
- Then select the Click event from the drop down box to the right
- Your empty code should be this (the code below has underscore characters added, so that it can fit on this page):

```
Private Sub mnuOpen_Click(ByVal sender As Object, _
ByVal e As System.EventArgs) _
Handles mnuOpen.Click


End Sub
```

With you cursor flashing between the two lines of your code, add the following:

```
openFD.ShowDialog()
```

When you typed a full stop after the **openFD**, you probably saw a list box appear. You can just double click the **ShowDialog()** item to add it to your code.

But this method of the OpenFileDialog control does what you'd expect it to do: Shows the dialogue box. You can even test it out right now. Press **F5** to run your programme. Then click the **Open** item on your File menu. You should see an **Open** dialogue box display.

Return to the design environment, and we'll explore some more things you can do with this Dialogue box control.

No more reading these lessons online - get the eBook here!

The Initial Directory

You can set which directory the dialogue box should display when it appears. Instead of it displaying the contents of the "My Documents" folder, for example, you can have it display the contents of any folder. This done with the Initial Directory property. Amend your code to this:

```
openFD.InitialDirectory = "C:\"
openFD.ShowDialog()
```

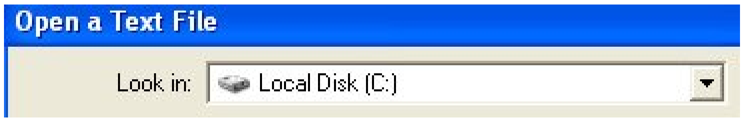
Run your programme again, and see the results in action. You should see the contents of the "C" folder on your hard drive (if you root folder is called something else, change the code above).

The Title Property

By default, the dialogue box will display the word "Open" as a caption at the top of your dialogue box. You can change this with the Title property. Add the line in Bold to your code:

```
openFD.InitialDirectory = "C:\"
openFD.Title = "Open a Text File"
openFD.ShowDialog()
```

Run your code again, and Click **File > Open** from your menu. You should see this at the top of the Open dialogue box:



In the next parts of this tutorial, we'll see how to change the Filter property, and how you can select a file from the list.

The Open File Dialogue Box Filter Property

This tutorial follows on from the previous section

In the previous section, we saw how to add an Open File Dialogue to our menus. We then saw how to add an Initial Directory and a Title property. In this section, we'll learn about the Filter property.

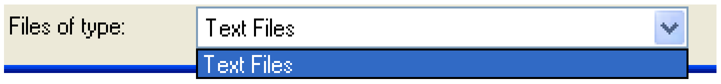
The Filter Property

In most dialogue boxes, you can display a list of specific files that can be opened. These are displayed in the "Files of Type" drop down list. To do this in VB.NET, you access the Filter property. We'll restrict our users to only opening Text files, those that end in the extension ".txt".

The following code (in bold) shows how to use the filter property:

```
openFD.InitialDirectory = "C:\"
openFD.Title = "Open a Text File"
openFD.Filter = "Text Files|*.txt"
openFD.ShowDialog()
```

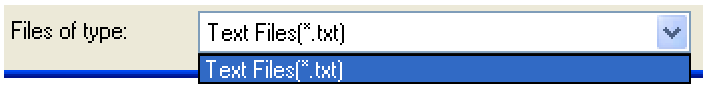
Run your code. Click **File > Open** on your menu, and then click the arrow on the drop down box for "Files of Type". You should see this:



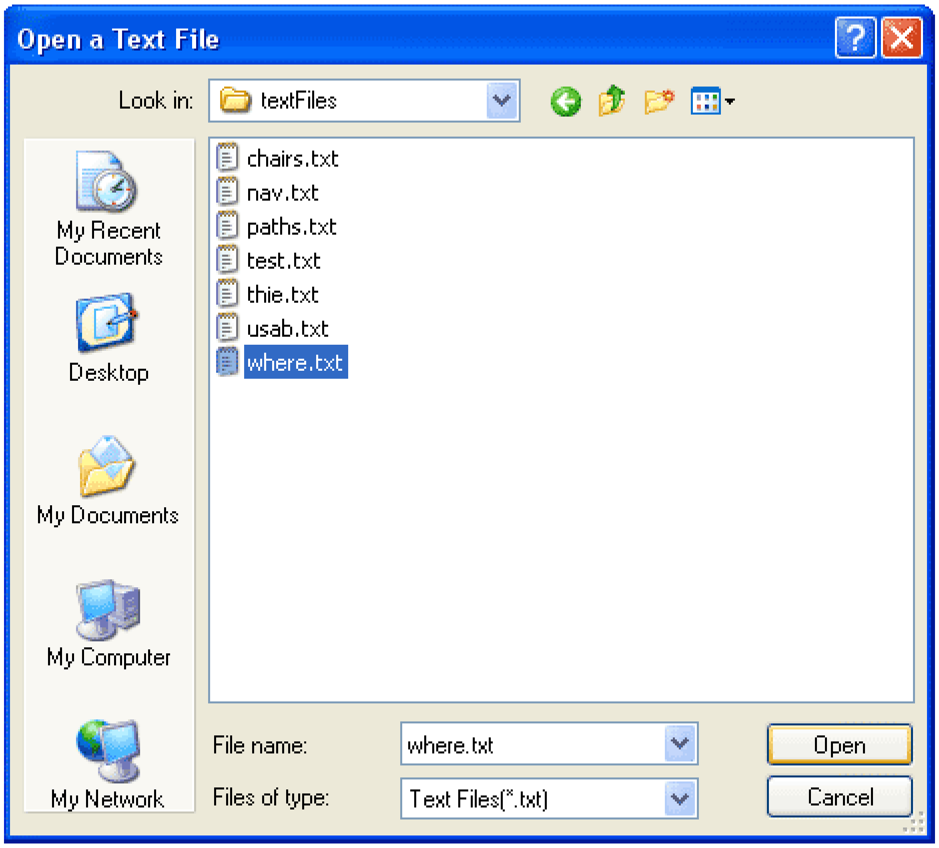
You can add a little bit extra to the description part of the filter, if you like. This will server a s a reminder of just what the extension is. Try amending the line to this:

openFD.Filter = "Text Files(*.txt)|*.txt"

When you run your code, you see this in the Files of Type area:



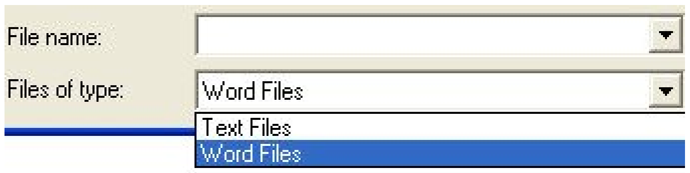
If you scroll across your Open dialogue box, you should see only text files displayed (you'll still see folders). If you can't see any files at all, double click a folder and explore. You'll soon see something like this:



To display files of more than one type, add a Pipe character between each filter. In the code below, two file types are specified, text files and Microsoft Word documents:

openFD.Filter = "Text Files|*.txt|Word Files|*.doc"

When the programme is run, you should be able to see two file types in the list:



In the next section, we'll see how to return which file was selected by the user.

Stay at Home and Learn

Select a File from the Open File Dialogue Box

This tutorial is part of an ongoing lesson. Click here for the first part.

You'll notice from the previous section that if you select a file and click the Open button, nothing happens. That's because the Open dialogue boxes doesn't actually open files! It only displays a list of files that CAN be opened, if you were clever enough to write the code. We'll be writing the code that does the opening (and the saving) in a later section. But you need to be able to get the name of the file. The Open Dialogue box has a property that returns the file name that was selected. Not surprisingly, it's called FileName:

```
OpenFD.FileName
```

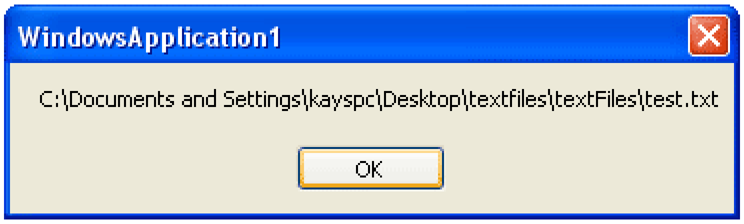
However, this is a property that returns a value (a string value). The value is the name of a file. So you have to assign this value to something. We can assign it to a new variable:

```
Dim strFileName As String

strFileName = OpenFD.FileName
```

The value in the variable **strFileName** will then hold the name of the file selected. So change you code to this (new lines in bold):

```
Dim strFileName As String

openFD.InitialDirectory = "C:\"
openFD.Title = "Open a Text File"
openFD.Filter = "Text Files|*.txt"
openFD.ShowDialog()
strFileName = OpenFD.FileName

MsgBox strFileName
```

Run your programme, and click your **File > Open** menu. Navigate to where you have some text files. Click one to select it. Then click the **Open** button. You should see the name of the file displayed in your message box:


WindowsApplication1

```
C:\Documents and Settings\kayspc\Desktop\textfiles\textFiles\test.txt
```

Notice that the location (the path) of the file is also displayed.

One thing you may have noticed is that if you select a file, then click the Cancel button, the message box still displays. But it will be blank. In your code, you will only want to do something with a file if the Cancel button is NOT clicked.

You can test to see if it was clicked by assigning the openFD.ShowDialog() to an integer:

```
Dim DidWork As Integer = openFD.ShowDialog()
```

You can then test what is inside of the DidWork variable. If the cancel button is clicked, the result of the action is stored by VB.NET in this property:

DialogResult.Cancel

You can compare the two in an if statement:

```
If DidWork = DialogResult.Cancel Then
MsgBox("Cancel Button Clicked")
Else
strFileName = openFD.FileName
MsgBox(strFileName)
End If
```

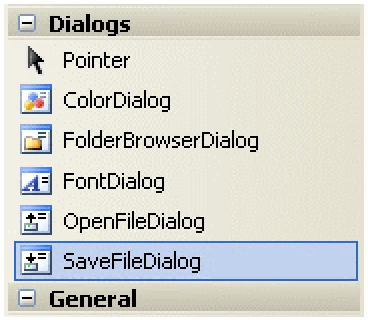
In the code above, you're only opening the file if the Open button was clicked. The code is a bit more complicated, but study it for a while and it will make sense!

In the next part, we'll take a look at how to code for the Save menu.

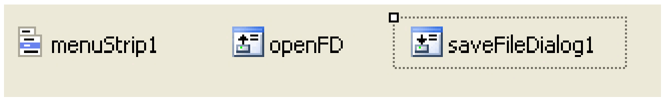
The SaveFileDialog Control

This tutorial is part of an ongoing lesson. Click here for the first part.

The save dialogue box works in the same way as the Open dialogue box. However, you can't use the same control. If you examine the Toolbox, you'll see a control called SaveFileDialog:



Double click this control to add one to your project. If you look at the bottom of the screen, you'll see the control added there, rather than onto your form:



In the image above, the control is selected. Changed the Name property of your control to something more manageable. Change it to **saveFD**. (You learned how to do this in a previous section.)

Access the code for your **File** > **Save** menu item. Then add the following code:

saveFD.ShowDialog()

Your code window should look like this one (Again, underscores have been added to the first line to fit on this page):

```
Private Sub mnuSave_Click(ByVal sender As Object, _
                          ByVal e As System.EventArgs) _
                          Handles mnuSave.Click

    saveFD.ShowDialog()

End Sub
```

Run your programme, then click your **File** > **Save** menu item. You should see the **Save As** dialogue box appear.

Just like the Open control, you can use the properties of the Save control on your dialogue boxes. Try changing these properties, just like you did with the Open properties:

Initial Directory
Title
Filter
FileName

There's another useful property you can use with the Save control - the Overwrite prompt. When you set this property, a message box pops up warning you that the file will be overwritten, and do you want to continue. To use this property, the code is this:

saveFD.OverwritePrompt = True

However, just like the Open box, when you click the Save button no file is actually being saved. You have to write your own code for this. You'll learn how to do this in a later section. For now, let's move on to the Edit menu.

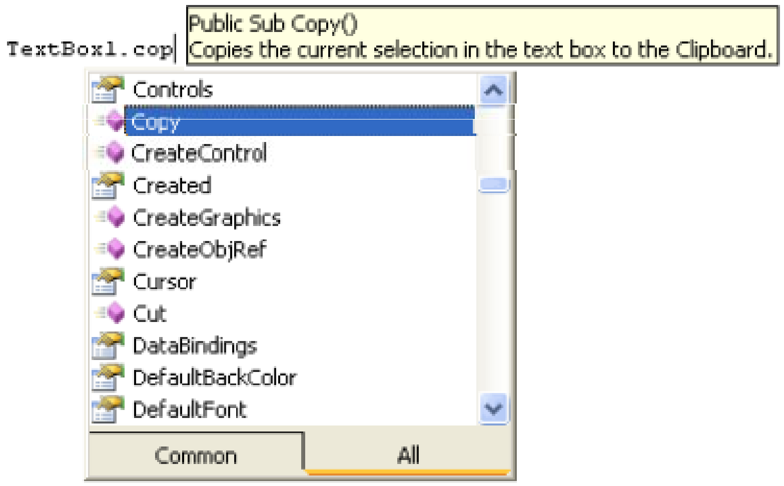
Cut, Copy, Paste and Undo in VB .NET

This tutorial is part of an ongoing lesson. Click here for the first part.

If you haven't already, add two textboxes to your form and set their MultiLine property to True. What we'll do now is to get the following menu items to work: **Undo**, **Cut**, **Copy** and **Paste**. We'll start with Copying:

The Copy Menu

If you type **Textbox1** in your code window, then a full stop, you get a list of properties and methods available to the textbox. Scroll up to the top and locate the Copy method:



```
TextBox1.cop
```

Public Sub Copy()
Copies the current selection in the text box to the Clipboard.

Notice the tool tip in yellow. It's telling you what this method does - copies the current selection to the clipboard. The clipboard is a temporary storage area available to most Windows applications. When you invoke the Copy method of the textbox, any selected text is place in this temporary storage area for you. You don't have to write any other code.

So access the code for your Copy menu item, and add this line to it:

Textbox1.Copy()

Your code window should look something like this:

```
Private Sub mnuCopy_Click(ByVal sender As Object, _
                          ByVal e As System.EventArgs) _
                          Handles mnuCopy.Click
    TextBox1.Copy()

End Sub
```

That's all there is for the copy menu! But nothing visible will happen when you run your code. Let's paste it into the second textbox.

No more reading these lessons online - get the eBook here!

The Paste Menu

Again, there's only one line of code to write. It's this:

TextBox2.Paste()

Notice that we're saying paste to textbox2. Because the copy menu places the text from textbox one onto the clipboard, you only need this one line of code. You're saying "Whatever is on the Clipboard, paste it into Textbox2".

So add that line to your Paste menu item. Your code window should look like this:

```
Private Sub mnuPaste_Click(ByVal sender As System.Object, _
                           ByVal e As System.EventArgs) _
                           Handles mnuPaste.Click
    TextBox2.Paste()

End Sub
```

Time to test it out. Run your programme. Select all the text in textbox one (it might already be selected), then click **Edit > Copy** from your menu.

Click inside the second textbox. Then click **Edit > Paste** from your menu. The text should appear in textbox two.

The Cut Menu

Access the code for you Cut menu item. Add the following code to it:

```
TextBox1.Cut()
```

Run your programme, and select the text in textbox one. From your menu, click **Edit > Cut**. The text should disappear (it's on the clipboard, though). Click inside textbox two, and click **Edit > Paste**. The text should be pasted over.

The Undo Menu

For the Undo menu, add this line of code:

```
TextBox1.Undo()
```

Run your programme. Then select the text in textbox one. Click **Edit > Cut** and the text disappears. Now click **Edit > Undo**. The text reappears.

The Edit menu we implemented is only a simple one. But it does demonstrate what you can do with VB.NET and menus.

We'll completed out look at menus by coding for the View menu you added to your form. In the process, we'll take a look at pictures boxes, as well as seeing how easy it is to hide and disable controls on a form.

How to Show and Hide Controls

This tutorial is part of an ongoing lesson. Click here if you haven't yet created a menu.

The items on our view menu are:

View Textboxes
View Labels
View Images

Although these are not terribly practical examples of what to place on a View menu, they will help us to demonstrate a few useful techniques. The first of these is how to show and hide controls.

The View Textboxes menu item

Controls on a form can be hidden or shown as the need arises. The process is quite easy. Access the code for your **View Textboxes** menu item. Type the following for the menu item:

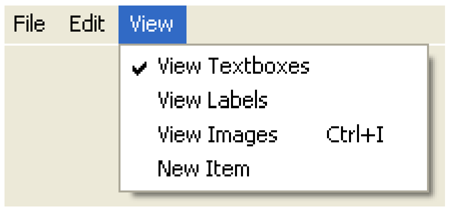
```
Textbox1.Visible = False
Textbox2.Visible = False
```

Run your code and test it out. Click **View > View Textboxes**. The two textboxes you added should disappear.

To hide a control, simply set it's Visible property to False. If you want to get it back, show a control by setting the Visible property to True.

A good idea is to have the ability to toggle a control on and off: One click of a menu item could hide the control, and a second click could show it again. You can do that with your menus.

Each item on your menu has a Checked property. If set to True, you'll see a tick appear next to the menu item. As in the image below:



You can use this Checked property as a toggle: If the menu item is ticked, display the textbox; if it's not ticked, hide the textbox.

Delete or comment out the line of code for your View Textboxes menu item. Add the following line in its place (this assumes that you've Named your View Textboxes menu item as mnuViewTextboxes. If you've named it something else, changed the part before the full stop):

```
mnuViewTextboxes.Checked = Not mnuViewTextboxes.Checked
```

This line toggles the Tick on and off. The part before the equals sign sets the Checked property of our menu item. The part after the equals sign sets it to whatever it's NOT at the moment. So if Checked is True, it's NOT False. In which case, set it to False.

Run your code and test it out. Click **View > View Textboxes**. Have a look at the menu again, and you'll see a tick appear. Click **View > View Textboxes** again and the tick will disappear.

We can show the textboxes if there's a tick next to View Textboxes. Just test the value of the Checked property in an If Statement. Add this If Statement just below your first line:

```
If mnuViewTextboxes.Checked = True Then
TextBox1.Visible = True
TextBox2.Visible = True
Else
TextBox1.Visible = False
TextBox2.Visible = False
End If
```

So the If Statement examines the Checked property of the menu item. If it's True, make the textboxes Visible ; Else, we set the Visible property of the textboxes to False.

Before you run your code, return to the Form view by holding Shift + F7 on your keyboard. When you have your form displayed, and not the code, click on textbox1 to select it. In the property box, locate the **Visible** property and set it to **False**. Do the same for texbox2. When your form runs, the two textboxes will then be hidden.

Now run your programme and test out your new menu. Click **View > View Textboxes** and see if they toggle on and off.

Exercise

Add two labels to your form. Write code to toggle the labels on and off. The two labels should disappear with the textboxes. And they should reappear when the menu item is toggled to the on position

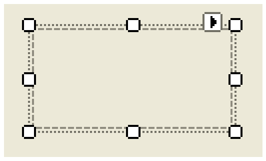
In the next part, we'll see how we can insert images. We'll use the View Images menu for that.

The View Images menu Item

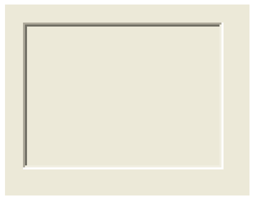
This tutorial is part of an ongoing lesson. Click here if you haven't yet created a menu.

Download the images you need for this tutorial

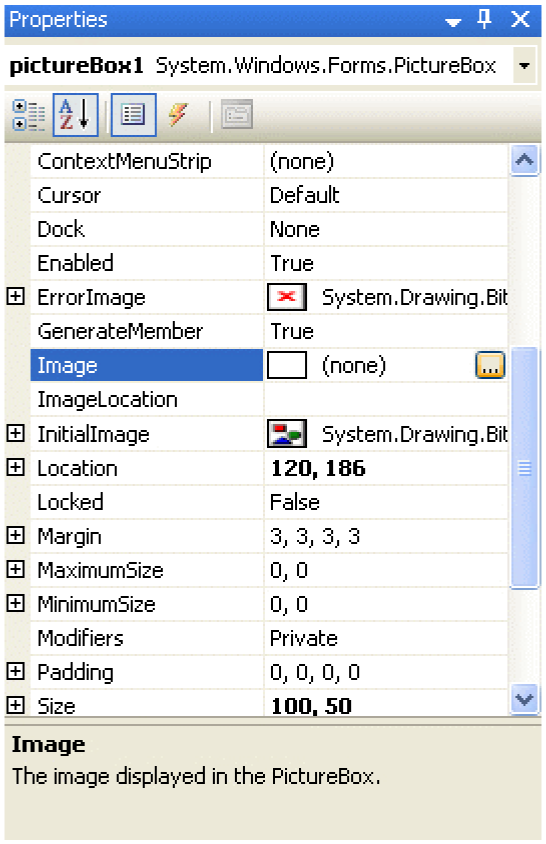
It's easy to add an image to your form with VB.Net. To insert an image, locate the **Picture** control in the toolbox. Either double click the control, or hold down your mouse on the form and draw one out. You should see something like this:



Change the **Height** and **Width** properties of the Picture Box to 100, 100. You'll have a small square. To make it stand out more, locate the **BorderStyle** property. Change the value to **Fixed3D**. Your Picture Box will then look like this:



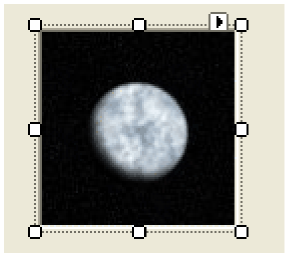
To add a picture at design time, locate the **Image** property in the properties box:



ContextMenuStrip	(none)
Cursor	Default
Dock	None
Enabled	True
⊞ ErrorImage	System.Drawing.Bit
GenerateMember	True
Image	(none)
ImageLocation	
⊞ InitialImage	System.Drawing.Bit
⊞ Location	**120, 186**
Locked	False
⊞ Margin	3, 3, 3, 3
⊞ MaximumSize	0, 0
⊞ MinimumSize	0, 0
Modifiers	Private
⊞ Padding	0, 0, 0, 0
⊞ Size	**100, 50**

Image
The image displayed in the PictureBox.

Download and unzip the image at the top of the page. Then click the button with the three dots on it. A dialogue box appears. Locate an image. Select it, and then click Open in the dialogue box. The image will appear in your Picture Box:



If you select an image that is too big for the picture box, only part if it will be visible. The Picture Box control does not resize your image.

You can, however, set another property of the picture box - the **SizeMode** property. Set this to **AutoSize** and your picture box will resize to the size of your image.

No more reading these lessons online - get the eBook here!

Insert an Image with your View Menu

You can use your open file dialogue box again to specify an image for the user to select. We'll do this from the View Images menu item.

Highlight you code for the **mnuOpen** item. (If you haven't yet coded for the **File > Open** menu item, click here.) Copy the first five lines, these lines:

```
Dim strFileName As String

openFD.InitialDirectory = "C:\"

openFD.Title = "Open an Text File"
openFD.Filter = "Text Files|*.txt"
Dim DidWork As Integer = openFD.ShowDialog()
```

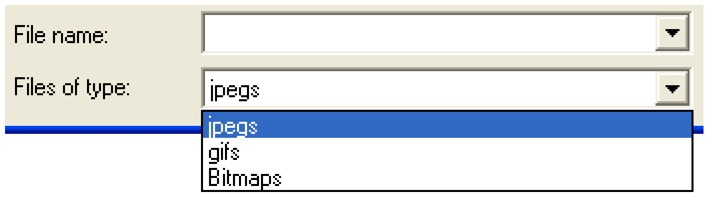
Paste them to your **mnuViewImages** menu item code. Change the Title property to this:

```
openFD.Title = "Open an Image"
```

And change the Filter property to this:

```
openFD.Filter = "jpegs|*.jpg|gifs|*.gif|Bitmaps|*.bmp"
```

Run your code and click your **View Images** menu item. You should see the Open dialogue box appear. If you look at the "Files of type" box, you should see this:



You should now be able to see only the three image formats we've specified.

To insert an image into your Picture Box, some new code is needed. Again though, we'll wrap it up in an If Statement.

Add the following code below the lines you've just added:

```
If DidWork <> DialogResult.Cancel Then
strFileName = openFD.FileName
PictureBox1.Image = Image.FromFile(strFileName)
openFD.Reset()
End If
```

There's only two lines you haven't met yet. The first is this line:

PictureBox1.Image = Image.FromFile(strFileName)

Previously, you were loading the image into the Image property of PictureBox1 directly from the Properties Box (by clicking the grey button with the three dots in it). Here, we're loading an image into the Image property using code. The way you do it is with the **FromFile** method of the Image Class.

Although that might be a bit baffling at this stage of your programming career, all it means is that there is some in-built code that allows you to load images from a file. In between round brackets, you type the name and path of the file you're trying to load. Since our file name has been placed inside of the **strFileName** variable, we can just use this. You can then assign this to the **Image** property of a Picture Box.

The last line, **openFD.Reset()**, will reset the initial directory of the open file dialogue box. To see what this does, comment out the line (put a single quote at the start of the line). Run your programme and **Click View > View Images**. Insert an image and then click **File > Open**. You'll notice that the files displayed in your dialogue are from the last directory you opened, rather than the one you set with "InitialDirectory = "C:\". By resetting the open dialogue box control, you're fixing this problem.

OK, that concludes our look at menus. We'll create a new programme now, and explore checkboxes, radio buttons and Group Boxes.

Check Boxes in VB .NET

Two more useful controls in the Visual Basic toolbox are the Check box and the Option Button. You use these when you want to give your users a choice of options. We'll add both of these to a new Form, and then combine them with a Select Case statement to read what the user has chosen.

Check Boxes

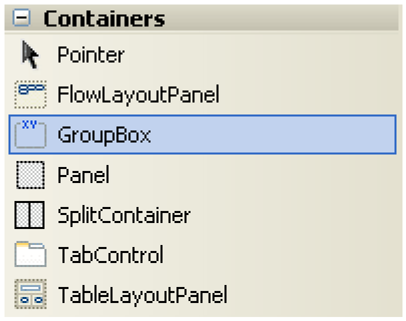
So start a new project. Locate the Checkbox control in the toolbox. Double click the control and a Checkbox appears on your new Form

You'll see that the Checkbox has the Text property of CheckBox1 by default, and a Name of CheckBox1. If you were to double click again on the Checkbox icon in the toolbox, the new control would be called CheckBox2.

The problem with this approach is that by double clicking each Checkbox, you have several individual Checkboxes. And if you wanted to move them around you'd have to move each Checkbox separately. There is a way to group all your Check Boxes together, and move them around as one - by using a **Group Box**. (You can use a Panel control as well, but we'll stick with the Group Box.)

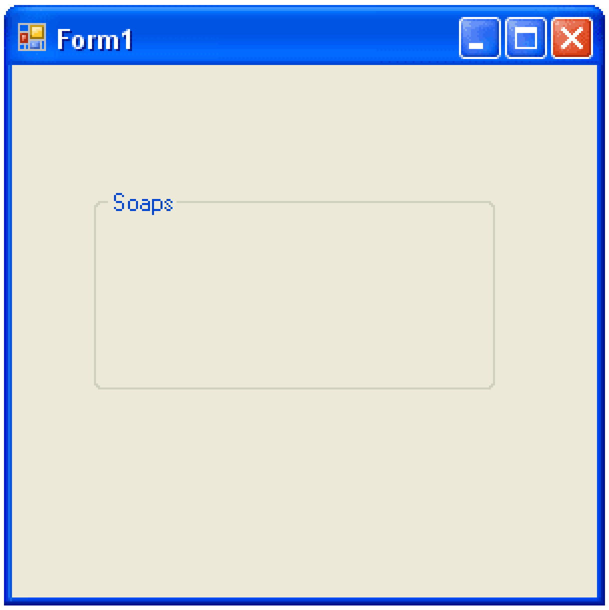
So, click on your Checkbox with the right mouse button. From the menu that pops up, select delete to get rid of it.

Now locate the **Group Box** control in the toolbox:



It's better to draw this one on the form, rather than dragging and dropping. When you've added one, the only thing you should have on your Form is a Group Box.

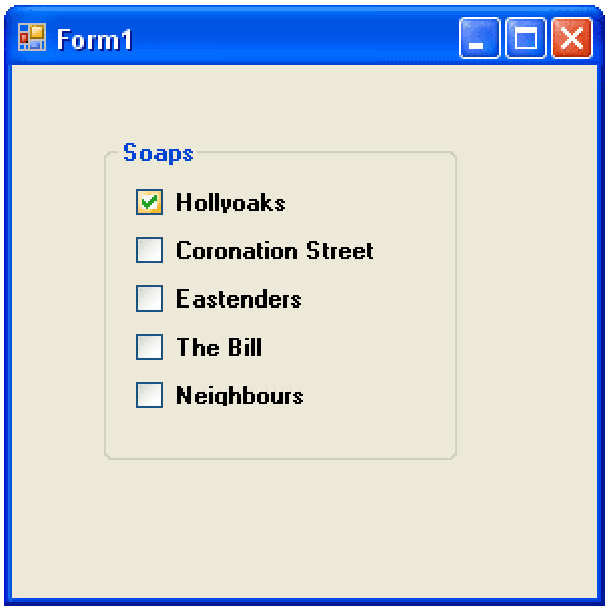
We're not going to be using many of the Properties in the Group Box Property box. But click on your Group Box to select it, and change to the Text Property to "Soaps". Change the Font Property to anything you like. You should now have a Form like this one



The Group Box we just added will hold our Checkboxes. It acts as a container for the controls. To move the Checkboxes about, we can just click on the Group Box to select it, and drag and drop the Group Box somewhere else. The Checkboxes will all move with the Group Box. Let's add some Checkboxes to it.

You CAN'T double click a checkbox and add it to a Group Box. The only way to add a control to a Group Box is to draw one on the Group Box.

1. Click once with your left mouse button on the **Checkbox** icon in the VB toolbox
2. Move your mouse pointer over to the inside of the **Group Box**. The mouse pointer will change to a cross
3. Hold down you left mouse button inside the Group Box. Keep the button held down, and drag outwards. Release the left button when you're happy with the size. You can always resize it later.
4. Add 5 Checkboxes to your Group Box
5. Change the **Text** property of each of your Checkboxes to any five Soap Operas. Your Form should now look something like the one below:



Run your programme to test it out. Click inside a Checkbox to select an item. Click again to deselect it. When you've finished, return to the Design Environment and click on the Group Box itself to select it. Make sure the Group Box IS selected, and not one of your Checkboxes. You can now drag the Group Box around your Form and all the Checkboxes will move with it.

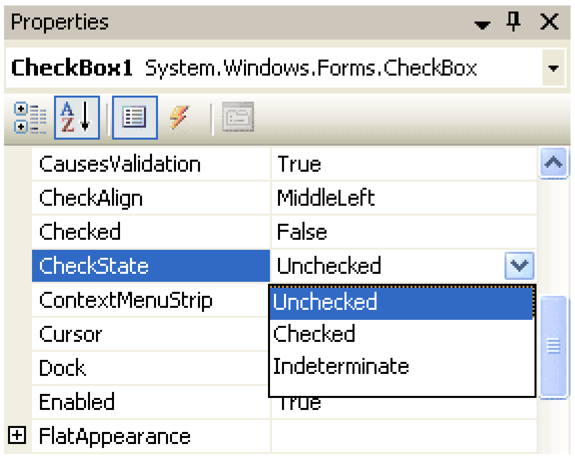
The point about having Checkboxes is to offer your users multiple choices. We'll now write some code to get the choices made by the user. All the Checkboxes with ticks inside them will have their Text displayed in a Message Box.

We'll do that in the next part of this tutorial.

The Checkbox Code

This lessons follows on from the previous page

If you click on any one of your Checkboxes and examine its Properties in the Property box, you'll notice that it has a **CheckState** Property. Click the down arrow to see the options this **CheckState** has.



CausesValidation	True
CheckAlign	MiddleLeft
Checked	False
CheckState	Unchecked
ContextMenuStrip	
Cursor	
Dock	
Enabled	True
FlatAppearance	

Unchecked
Checked
Indeterminate

As you can see, you are given three options: Unchecked, Checked, Indeterminate.

If a checkbox has been selected, the value for the CheckState property will be 1; if it hasn't been selected, the value is zero. (The value for the Indeterminate option is also zero, but we won't be using this.)

We're only going to test for 0 or 1, Checked or Unchecked. You can do the testing with a simple If Statement. Like this:

```
If CheckBox1.CheckState = 1 Then

MsgBox("Checked")

End If
```

After you type the equal sign, though, VB will give you a drop down box of the values you can choose from. So the above code is the same as this:

```
If CheckBox1.CheckState = CheckState.Checked Then

MsgBox("Checked")

End If
```

Whichever you choose, the Statement will be True if the checkbox is ticked and False if it isn't.

Add a Button to your Form and put that code behind it (either of the two, or test both). When you've finished typing the code, run your programme. Put a tick inside Checkbox1, and click your button. You should get a Message Box popping up.

Amend your code to this:

```
If CheckBox1.CheckState = CheckState.Checked Then
```

```
MsgBox("Checked")
Else

MsgBox("Not Checked")

End If
```

An alternative to Else is ElseIf. It works like this:

```
If CheckBox1. CheckState = 1 Then

MsgBox "Checked"

ElseIf Checkbox1. CheckState = 0 Then

MsgBox "Unchecked"

End If
```

When using the ElseIf clause, you need to put what you are testing for on the same line, just after ElseIf. You put the word Then right at the end of the line. You can have as many ElseIf clauses as you want. In other words, it's exactly the same as the first "If" line only with the word "Else" in front "If".

No more reading these lessons online - get the eBook here!

Add 4 more If Statements to check for the values in your other Checkboxes :

Checkbox2.CheckState, Checkbox3.CheckState, etc.

We're now going to get rid of the Message Boxes inside the If Statements. So either comment out all your MsgBox lines, or delete them altogether.

Instead, we'll build up a String Variable. So add a String Variable to the code for your button, and call it message.

The message variable needs to go inside the If Statement. If the user has checked a Box (If its CheckState property is 1), then we build the message. We need to remember what is inside the message variable, so we can just use this:

```
message = message & Checkbox1.Text & vbNewLine
```

That way, every time an option is Checked, Visual Basic will keep what is in the variable called message and add to it whatever the text is for the Checkbox.

So add that line to your If Statements. Something like this

```
If Checkbox1.CheckState = 1 Then

message = message & Checkbox1.Text & vbNewLine

End If

If Checkbox2.CheckState = 1 Then

message = message & Checkbox2.Text & vbNewLine

End If
```

And at the end of your If Statements, on a new line, add this:

```
MsgBox "You have chosen " & vbNewLine & message
```

Here, we're building a text message for out Message Box. We're saying our Message Box is to contain the text "You have chosen " And a New Line And whatever is inside the variable called message.

When you've finished, Run your Programme to test it out. Put a tick inside all of your Checkboxes. When you click your button all your Soap Operas should appear in the Message Box. Check and Uncheck some options, and click the button again. Only those items that are selected should appear in your Checkbox.

So, we can test to see which Check Boxes a user has ticked, and we can keep a record of those choices for ourselves.

What we can also do is count how many Check Boxes were ticked. We can then use a Select Case Statement to display a suitable message.

Keeping a count is straightforward. First we set up an integer variable called counter, and set it's value to zero.

```
Dim counter As Integer = 0
```

Then we can just keep adding one to whatever is in this counter variable. In other words, every time a Checkbox has a value of 1 (is ticked), we can add one to our counter (increment our variable).

We can add this straight into our If Statement, on a new line after the message code.

```
counter = counter + 1
```

So your code would be this:

```
If Checkbox1.CheckState = 1 Then

message = message & Checkbox1.Text & vbNewLine
counter = counter + 1

End If
```

To test that your counter is working, you can add a second message box to the end of the code, just below your first message box:

```
MsgBox("Counter = " & counter)
```

Or adapt your first message box:

```
MsgBox("You have chosen " & counter & " soaps")
```

Now that we have a way to count how many Checkboxes are ticked, we can add a Select Case Statement.

Exercise

Add a Select Case Statement to the end of your code to test whatever is inside the variable called counter.

Remember what the format is for a Select Case? It's this:

```
Select Case VariableName
Case 0
MsgBox "You obviously don't watch soaps!"
End Select
```

If you have 5 Check Boxes on your Form, then the maximum value that counter will hold is 5. So you only need to go up to Case 5.

Add suitable messages for each Case that you're testing for.

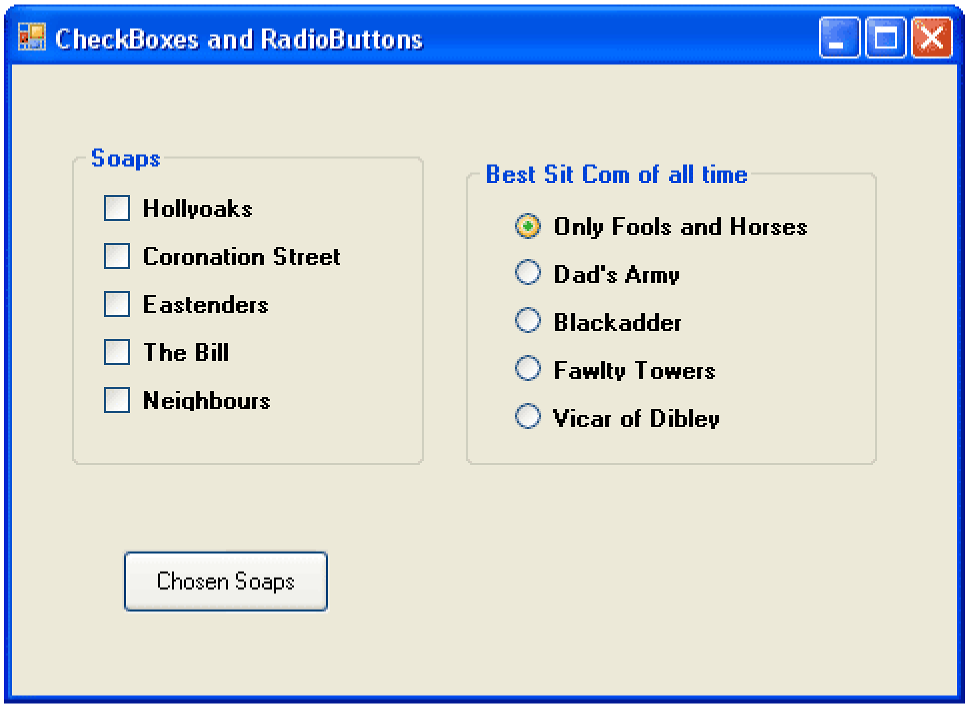
This has been quite a long lesson, so give yourself a big pat on theback if you got through it! In the next part, we'll see how to add Option Buttons to a VB .NET form.

Add Option Buttons to a VB .NET form

Radio Buttons, sometimes called Option Buttons, are used when you want to restrict a user's choice to one, Male/Female, for example. A Checkbox would be no good here, because a user could tick both boxes. You want to force your users to pick only one from your list of options.

Adding Radio Buttons to a Form is exactly the same process as adding a Checkbox. Again, we'll add some Radio Buttons to a Group Box, and write code to extract what the user has chosen.

1. Add a **Group Box** to your Form.
2. Set the **Text** Property of the Group Box to "Best Sit Com of all time"
3. Set the **Font** options to anything you like
4. Place five **Radio Buttons** into your Group Box (By default, they'll be called "Option1", "Option2", "Option3", etc
5. Set the Text Property of the Five Radio Buttons to Only Fools and Horses, Dad's Army, Blackadder, Fawlty Towers, Vicar of Dibley
6. Your Form should now look something like this:



Run your programme and test to see if you can indeed only select one item from the list.

The reason you can only select one is that all the radio buttons are placed in the same group box. You can place another set of radio buttons in a second group box, and these would work independently of the set of radio buttons in the first group box.

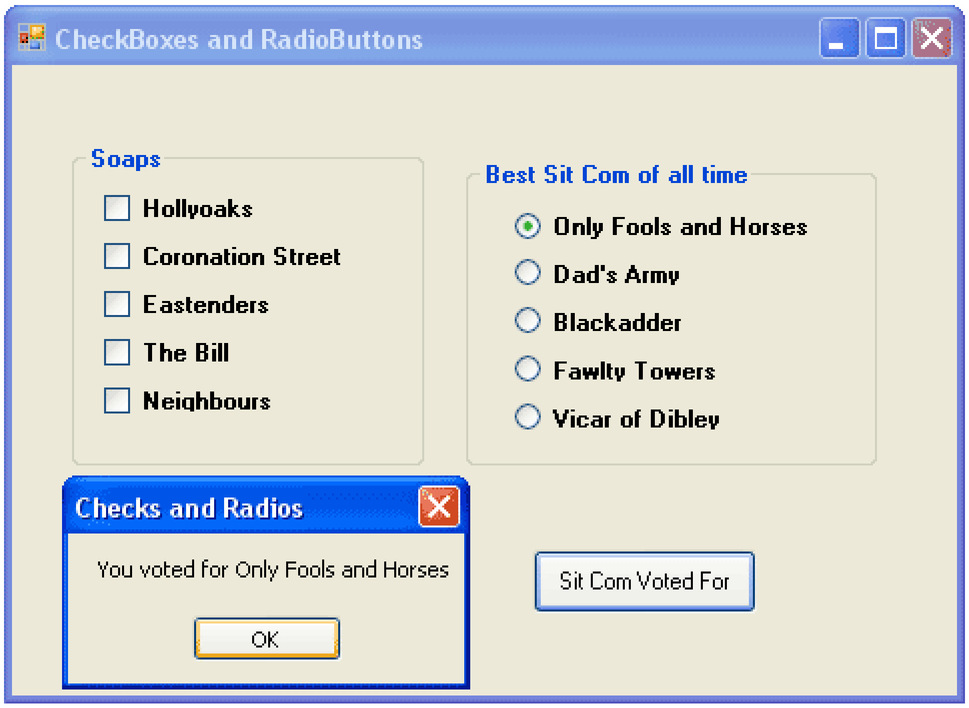
To test which Sit Com was chosen, you can use an If … Elseif Statement. You can do this because only one of the radio buttons will be True if selected: all the others will then have a value of False.

So place a Button on your form. Set the Text property to something appropriate. Then double click your new button to open up the code window. Type the following code (Notice that the Property is now Checked, and not CheckState):

```
Dim ChosenSitCom As String

If RadioButton1.Checked = True Then
ChosenSitCom = RadioButton1.Text
ElseIf RadioButton2.Checked = True Then
ChosenSitCom = RadioButton2.Text
ElseIf RadioButton3.Checked = True Then
ChosenSitCom = RadioButton3.Text
ElseIf RadioButton4.Checked = True Then
ChosenSitCom = RadioButton4.Text
ElseIf RadioButton5.Checked = True Then
ChosenSitCom = RadioButton5.Text
End If


MsgBox("You voted for " & ChosenSitCom)
```

By using If … ElseIf we can check which radio button a user selected. The Text property from the chosen radio button is then placed in a String variable called ChosenSitCom. At the end, we then display the selected radio button in a message box.

Run your programme and test it out. Select a Sit Com, and then click your Button. You should see the item you selected displayed:



You voted for Only Fools and Horses

Exercise

Add a Textbox to your Form. Write code to transfer a chosen Sit Com to the Textbox when the button is clicked. Add a label next to the Textbox with the Caption "You Voted For. . . "

And that's all there is to adding Option Buttons to your VB .NET forms. In the next section of the course, we'll take a look at error checking.

Error Handling and Debugging in VB .NET

Debugging your code is something you will need to do. Unless you write perfect code every time, there's no getting away from it. In this section, we'll take a look at ways you can track down errors using VB.NET.

Types of Error

Programming errors are generally broken down into three types: **Design-time**, **Runtime**, and **Logic** errors.

A **Design-time error** is also known as a syntax error. These occur when the environment you're programming in doesn't understand your code. These are easy to track down in VB.NET, because you get a blue wiggly line pointing them out. If you try to run the programme, you'll get a dialogue box popping up telling you that there were Build errors.

Runtime errors are a lot harder to track down. As their name suggests, these errors occur when the programme is running. They happen when your programme tries to do something it shouldn't be doing. An example is trying to access a file that doesn't exist. Runtime errors usually cause your programme to crash. If and when that happens, you get the blame. After all, you're the programmer, and you should write code to trap runtime errors. If you're trying to open a database in a specific location, and the database has been moved, a Runtime error will occur. It's your job to predict a thing like this, and code accordingly.

Logic errors also occur when the programme is running. They happen when your code doesn't quite behave the way you thought it would. A classic example is creating an infinite loop of the type "Do While x is greater than 10". If x is always going to be greater than 10, then the loop has no way to exit, and just keeps going round and round. Logic errors tend not to crash your programme. But they will ensure that it doesn't work properly.

In the next few pages, we'll take a closer look at all three types of error.

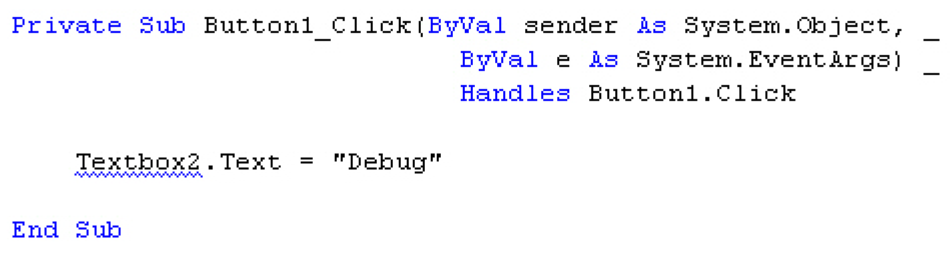
Design Time Errors in VB .NET

Design time errors, remember, are syntax errors. It's when VB .NET hasn't understood what you have typed. If that isn't clear, a few examples should explain it beter. So do the following:

- Create a new Windows project
- Add a **button** and **textbox** to your form
- Leave the **Name** properties on the defaults of **Button1** and **Textbox1**
- Double click your button to access its code, and type the following:

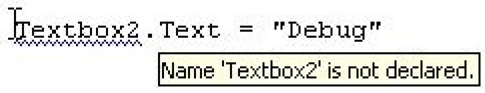
Textbox2.Text = "Debug"

When you finish typing the line, VB.NET puts a blue wiggly line under Textbox2:



```
Private Sub Button1_Click(ByVal sender As System.Object, _
                           ByVal e As System.EventArgs) _
                           Handles Button1.Click

    Textbox2.Text = "Debug"

End Sub
```

If you hold your mouse over **Textbox2**, you'll see a yellow tool tip appear, like the following:



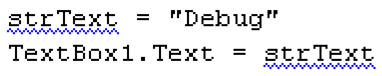
```
Textbox2.Text = "Debug"
```

Name 'Textbox2' is not declared.

The error is occurring because you don't have a textbox called Textbox2. You'll also see this same "**Not declared**" error if you try to set up a variable on the fly (which you were allowed to do in previous version of VB.) As an example, change your code to this:

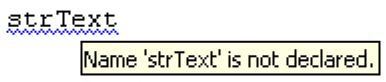
strText = "Debug"
TextBox1.Text = strText

Here, we're trying to put the word "Debug" into a variable called strText. We then want to assign this variable to the Text property of Textbox1. However, VB.NET protests about this, and puts a wiggly line under all occurrences of strText:



```
strText = "Debug"
TextBox1.Text = strText
```

Hold your mouse over the variable strText and you'll see the "not declared" tip again:



```
strText
```

Name 'strText' is not declared.

The problem this time is that we haven't declared the variable strText. Change the code to this:

Dim strText As String

strText = "Debug"
TextBox1.Text = strText

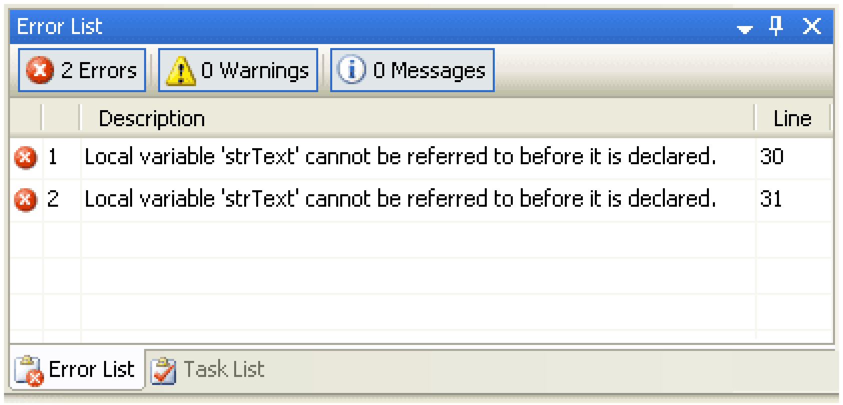
Now that we have declared a variable, the wiggly lines will go away. If we added the variable declaration in the wrong place, however, the wiggly lines would come back. Change you code to this:

strText = "Debug"
TextBox1.Text = strText

Dim strText As String

The wiggly lines will be back. That's because the declaration comes on the third line. When VB.NET meets the first two lines, it knows nothing about the strText variable.

If you have the Task List window open, you'll see a report of your error (If you can't see the Task List window, from the menu bars click **View** > **Other Windows** > **Task List**. Or hold down the **Ctrl** and the **Alt** keys on your keyboard, and the press the letter **K**.):



	Description	Line
1	Local variable 'strText' cannot be referred to before it is declared.	30
2	Local variable 'strText' cannot be referred to before it is declared.	31

The description of the error is "**Local variable 'strText' cannot be referred to before it is declared**". If you double click the icons on the left, VB.NET will highlight the error in your code.

Move the "Dim … " Line back to the top, and not only do the blue wiggly lines go away, but the Task List will be erased.

Design-time errors like the one above can be quite straightforward to correct. Others can be quite difficult, and you'll see the blue wiggly line but not understand how to correct the error. The Task List should be your first port of call when faced with such an error.

In the next part, we'll take a look at Runtime errors.

Runtime errors in VB .NET

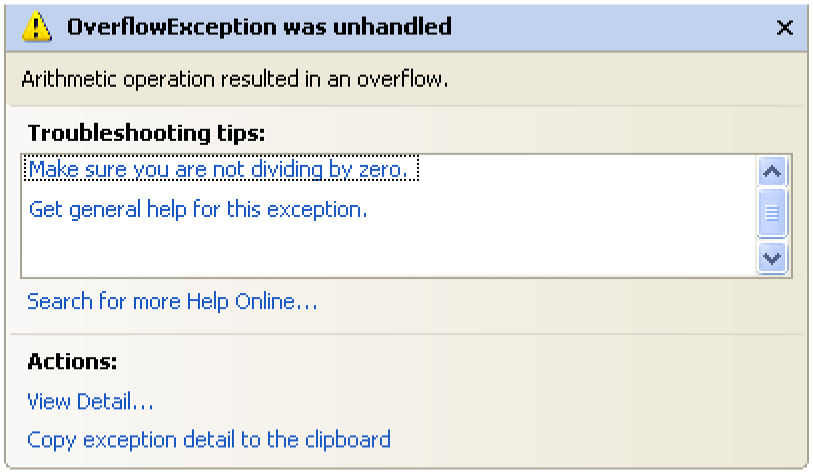
This lesson is part of an ongoing tutorial. The first part is here: Design Time Errors

Runtime errors are a lot harder than Design Time errors to track down. As their name suggests, these errors occur when the programme is running. Runtime errors are the ones that crash your programme. A simple way to crash a programme is to divided by zero. Change the code for your button to this, and try it out:

```
Dim Num1 As Integer
Dim Num2 As Integer

Num1 = 10
Num2 = 0

TextBox1.Text = CInt(Num1 / Num2)
```

The **CInt( )** part means Convert to an Integer. We're just making sure to convert the answer to the sum into a number. But run your programme and test it out. Click your button and see what happens.

What happens is that you'll get the following error message popping up:



OverflowException was unhandled

Arithmetic operation resulted in an overflow.

Troubleshooting tips:
Make sure you are not dividing by zero.
Get general help for this exception.
Search for more Help Online...
Actions:
View Detail...
Copy exception detail to the clipboard

Click the Break button, and then stop your programme from running.

When you try to divide by zero, VB.NET throws up the **Overflow** error message - there would be just too many zeros to go into the **Integer** variable type. Even if you change the Type into a **Single** or a **Double**, you'd still get the same error message. Programming environments just don't like you dividing a number by zero. If this were in a real programme, chances are it would crash, or "bug out". And you'll get the blame!

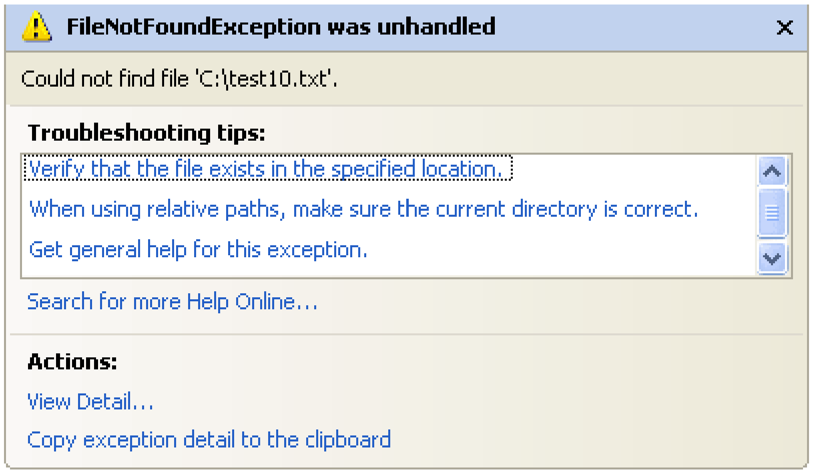
If you think the answer to a calculation could result in zero, you should check for this. We'll see how to write code to trap Runtime errors in a moment. But here's another example of one.

From the controls toolbox, add a RichTextBox control to your form. Change the **Name** property of your **RichTextBox** to **rt1**. A RichTextBox is just like a normal textbox but with more functionality. One of these extra functions is the ability to load a file directly. Delete or comment out any code you have for your button, and add the following line:

```
rt1.LoadFile("C:\test10.txt", RichTextBoxStreamType.PlainText)
```

All the line does is to load (or try to) the text file called "**test10.txt**" into the RichTextBox. The second argument just specifies that the type of file we want to load is a Plain Text file.

Run your programme, and then click the button. If you don't have a text file called "test10.txt" in the root folder of your C drive, you'll get the following Runtime error message:



FileNotFoundException was unhandled

Could not find file 'C:\test10.txt'.

Troubleshooting tips:

Verify that the file exists in the specified location.

When using relative paths, make sure the current directory is correct.

Get general help for this exception.

Search for more Help Online...

Actions:

View Detail...

Copy exception detail to the clipboard

The additional information is quite useful this time. It's saying that the file "C:\test10.txt" could not be found. If the error occurred in a normal programme, it would shut down. Not something you want a programme to do in mid stream! But let's see how to deal with it.

In the next part we take a look at Try ... Catch in order to trap any errors in your code.

Try … Catch in VB .NET

This lesson is part of an ongoing tutorial. The previous part is here: Runtime Errors

VB.NET has a inbuilt class that deals with errors. The Class is called **Exception**. When an exception error is found, an **Exception object** is created. The coding structure VB.NET uses to deal with such Exceptions is called the Try … Catch structure.

In the coding area for your button, type the word **Try**. Then hit the return key on your keyboard. VB.NET completes the rest of the structure for you:

```
Try

Catch ex As Exception

End Try
```

The Try word means "Try to execute this code". The Catch word means "Catch any errors here". The ex is a variable, and the type of variable it is is an Exception object.

Move your line of code from the previous section to the Try part:

```
Try
```

rt1.LoadFile("C:\test10.txt", RichTextBoxStreamType.PlainText)

Catch ex As Exception

End Try

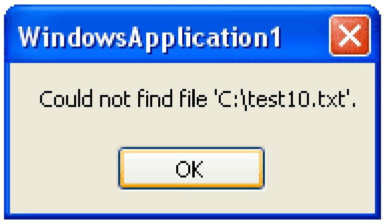
When you run your programme, VB will Try to execute any code in the Try part. If everything goes well, then it skips the Catch part. However, if an error occurs, VB.NET jumps straight to Catch. Add the following to your Catch part:

MsgBox(ex.Message)

Your coding window should look like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
                          Handles Button1.Click

    Try

        rt1.LoadFile("C:\test10.txt", RichTextBoxStreamType.PlainText)

    Catch ex As Exception

        MsgBox(ex.Message)

    End Try

End Sub
```

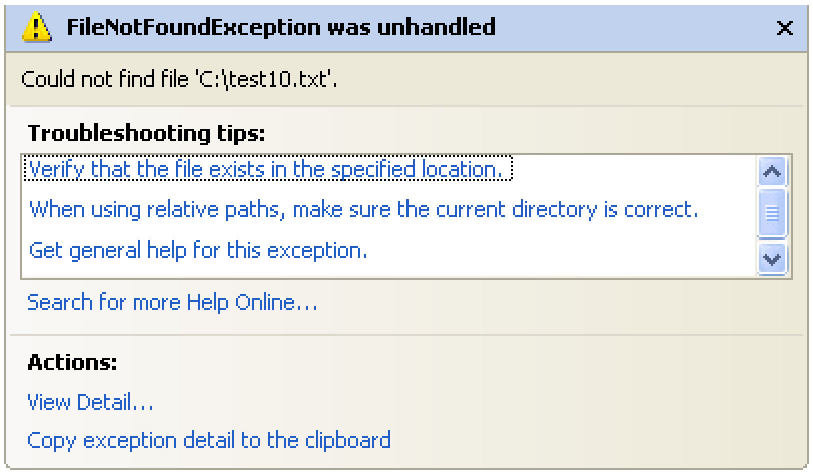
Because ex is an object variable, it now has its own Properties and methods. One of these is the Message property. Run your programme and test it out. Click your button. You should see the following error message:



Could not find file 'C:\test10.txt'.

The message is coming from the "additional Information" section of the error message we saw earlier, the one we didn't handle. But the point about this new message box is that it will not crash your programme. You have handled the Exception, and displayed an appropriate message for the user.

If you know the kind of error that a programme might throw, you can get what Type it is from the Error message box you saw earlier. This one:



FileNotFoundException was unhandled

Could not find file 'C:\test10.txt'.

Troubleshooting tips:

Verify that the file exists in the specified location.

When using relative paths, make sure the current directory is correct.

Get general help for this exception.

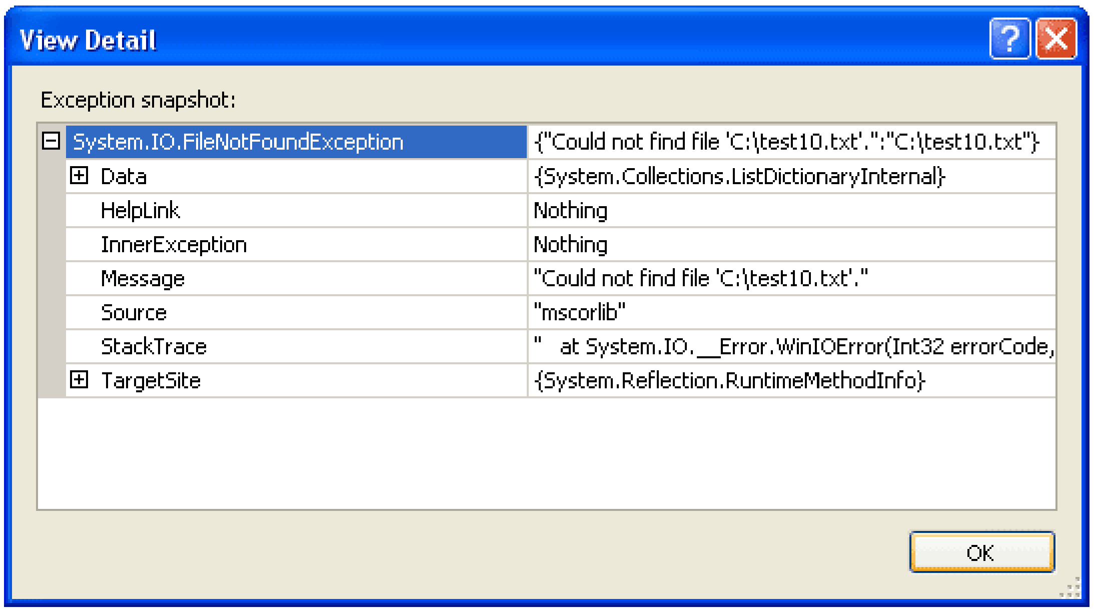
Search for more Help Online...

Actions:

View Detail...

Copy exception detail to the clipboard

Click the **View Details** links under **Actions** to see the following:



View Detail

Exception snapshot:

System.IO.FileNotFoundException	{"Could not find file 'C:\test10.txt'.":"C:\test10.txt"}
Data	{System.Collections.ListDictionaryInternal}
HelpLink	Nothing
InnerException	Nothing
Message	"Could not find file 'C:\test10.txt'."
Source	"mscorlib"
StackTrace	" at System.IO.__Error.WinIOError(Int32 errorCode,
TargetSite	{System.Reflection.RuntimeMethodInfo}

The first line tells us the Type of Exception it is:

System.IO.FileNotFoundException

You can add this directly to the catch part. Previously, you were just catching any error that might be thrown:

Catch ex As Exception

But if you know a "file not found" error might be thrown, you can add that to the Catch line, instead of Exception:

```
Catch ex As System.IO.FileNotFoundException
```

You can keep the Exception line as well. (You can have as many Catch parts as you want.) This will Catch any other errors that may occur:

```
Try

rt1.LoadFile("C:\test10.txt", RichTextBoxStreamType.PlainText)

Catch ex As System.IO.FileNotFoundException

MsgBox(ex.Message)

Catch ex As Exception

MsgBox(ex.Message)

End Try
```

There is one last part of the Try … Catch Statement that VB.NET doesn't add for you - **Finally**:

```
Try

Catch ex As Exception

Finally

End Try
```

The **Finally** part is always executed, whether an error occurs or not. You typically add a Finally part to perform any cleanup operations that are needed. For example, you may have opened a file before going into a Try … Catch Statement. If an error occurs, the file will still be open. Whether an error occurs or not, you still need to close the file. You can do that in the **Finally** part.

But Microsoft advise that you always use **Try … Catch** Statements in your code. However, throughout the rest of this course, for convenience sake, we won't be using them much. Even when we should be.But that's no excuse for you not to use them!

In the next part, we'll take a look at Logic Errors.

Logic Errors in VB .NET

This lesson is part of an ongoing tutorial. The first part is here: Design Time Errors

The third category of errors are the Logic errors. These can be thought of as coding errors. Your coding errors. They can be quite tricky to track down, and have you tearing your hair out with frustration. You will often hear yourself saying "But that should work! Why won't it!"

Add another button to the form tou created in the first part, and try this code as an example of a logic error:

```
Dim x As Integer
Dim y As Integer
Dim answer As Integer

x = 10.5
y = 3
answer = x * y
TextBox1.Text = answer
```

When you've added the code to your button, run your programme and test it out. Before you click the button, what answer did you expect to get?

You'd think that 10.5 multiplied by 3 would give you the answer 31.5. Click your button. The answer that appears in your textbox is 30!

This is a logic error: when you don't get the answer you thought you'd get. The problem, if you were paying attention during the variable types sections, is that we are trying to put floating point numbers into an Integer variable type. The Integer variable only works with whole numbers. When you assign 10.5 to the variable x, the point 5 on the end gets chopped off. So only the 10 gets stored in x. 10 times 3 is thirty, and this is the answer that appears in the textbox.

But the point is that VB.NET did not raise a Design-time error. Nor did it raise a Runtime error. The programme executed, and did not "bug out" on us. It just didn't give you the answer you expected - it was a logic error.

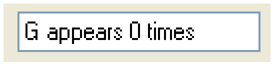
Logic errors can be fairly simple to track down and solve. (The problem above can be solved by changing the variable types from Integer to Single or Double.) But they can also be quite difficult to track down. Especially as your code gets longer and longer. Here's another example of a logic error.

Erase the code you have for button2, and add the following instead:

```
Dim i As Integer
Dim LetterCount As Integer
Dim strText As String
Dim letter As Char
```

```
strText = "Debugging"

For i = 1 To strText.Length - 1
letter = strText.Substring(1)

If letter = "g" Then
LetterCount = LetterCount + 1
End If
Next

TextBox1.Text = "G appears " & LetterCount & " times"
```

All the code does is to try and count how many times the letter "g" appears in the word "Debugging". We're using a For loop, and Substring to get one letter at a time. This single letter is then placed inside the variable called letter. An If Statement is used to check if the letter is a "g". If it is, we increment the LetterCount variable. The answer we're expecting in the textbox is 3. Except, we don't get 3. We get zero:



```
G appears 0 times
```

There were no wiggly lines and therefore no Build errors. When the button was clicked, a Runtime exception did not crash the programme. So that leaves a logic error. But where is it?

In the next part, you'll learn how to use VB .NET's inbuilt tools to help you track down logic errors.

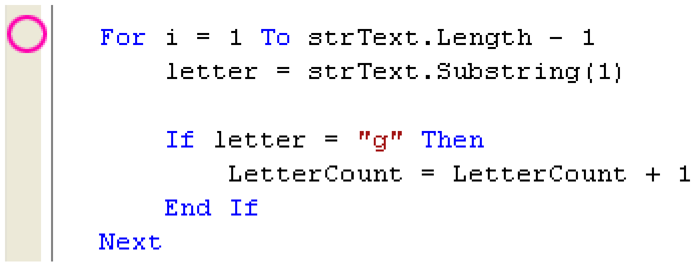
Breakpoints and Debugging Tools

This lesson is part of an ongoing tutorial. The first part is here: Design Time Errors

In the last part, you added some code to a button. All the code does is try and count how many times the letter "g" appeared in the word "Debugging". We saw that the programme failed to do this, and came up with the answer zero. To help you find out what went wrong, there is a tool in VB .NET called a Breakpoint. Let's see what this is, and how to use Breakpoints.

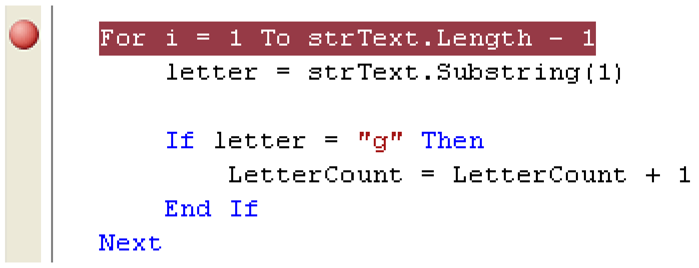
Breakpoints

A breakpoint is like a note to VB.NET to stop your programme at a particular place. You add one to your code by clicking in the margins. A brown circled then appears, indicating where the code will break. The following two images show how to add one:



```
For i = 1 To strText.Length - 1
    letter = strText.Substring(1)

    If letter = "g" Then
        LetterCount = LetterCount + 1
    End If
Next
```

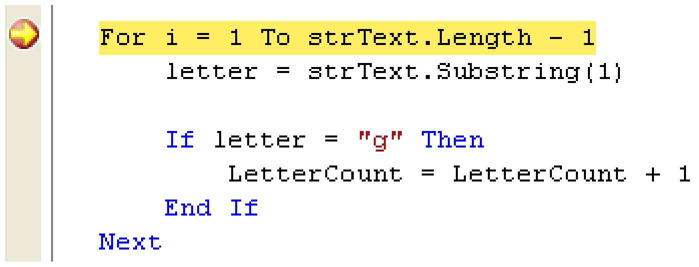
When you click in the margins, to the left of a line of code, a brown circle appears:



```
For i = 1 To strText.Length - 1
    letter = strText.Substring(1)

    If letter = "g" Then
        LetterCount = LetterCount + 1
    End If
Next
```

Notice that the line where you want VB.NET to break is highlighted brown.

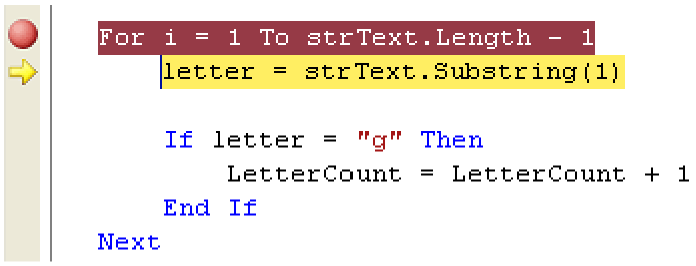
Run your programme, and click the button. You are immediately returned to the coding window. The place where you put the Breakpoint will now have a yellow arrow on top of the brown circle. The brown highlighted line will now be yellow:



```
For i = 1 To strText.Length - 1
    letter = strText.Substring(1)

    If letter = "g" Then
        LetterCount = LetterCount + 1
    End If
Next
```

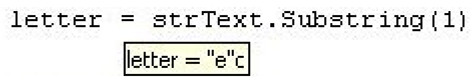
The yellow highlight indicates where in your code VB.NET is. To continue checking your code, press F10 on your keyboard (you can also press F11, but this will jump into any Subs or Functions you've set up.)

The next line in your code will be highlighted:



```
For i = 1 To strText.Length - 1
    letter = strText.Substring(1)

    If letter = "g" Then
        LetterCount = LetterCount + 1
    End If
Next
```

The yellow arrow, and the yellow highlight, jump down one line. Press the F10 key again. Then hold you mouse on the letter variable. The value this variable currently holds will be displayed:



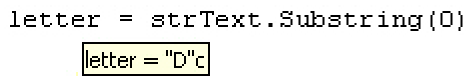
```
letter = strText.Substring(1)
        letter = "e"c
```

The first time round the loop, the value in **letter** is "e" (The "c" next to it means that the variable type is Character).

If the "e" of "Debugging" is getting checked first, what happened to the "D"? Straight away, this indicates a problem. And the problem is that the Substring method starts counting from zero. So halt your programme by clicking "**Debug > Stop Debugging**", or click the **Stop** icon on the toolbar. Change the line in question to this:

letter = strText.Substring(0)

Run your programme again, and click the button. When you are returned to your code, press the F10 key and check the value of the letter variable. It should now be this:



```
letter = strText.Substring(0)
      letter = "D"c
```

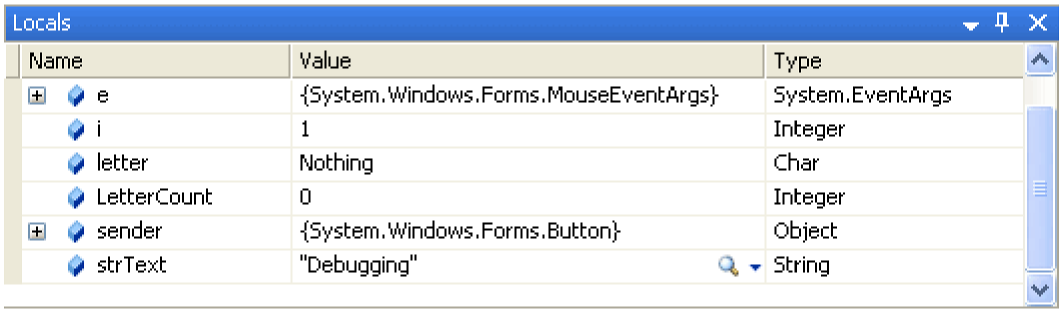
This time, the code is catching the first letter of the word when the loop begins, and not the second one.

Is that it? Have we found the cause of our problems? Stop your programme. Click on the brown circle to get rid of the Breakpoint. Run it again, and see what happens.

The number of G's counted is still zero! So the logic error has not yet been tracked down. Create another Breakpoint at the same place, and try again.

You can continue pressing the F10 key until you've spotted the error. Or you can use another debugging tool - the Locals window.

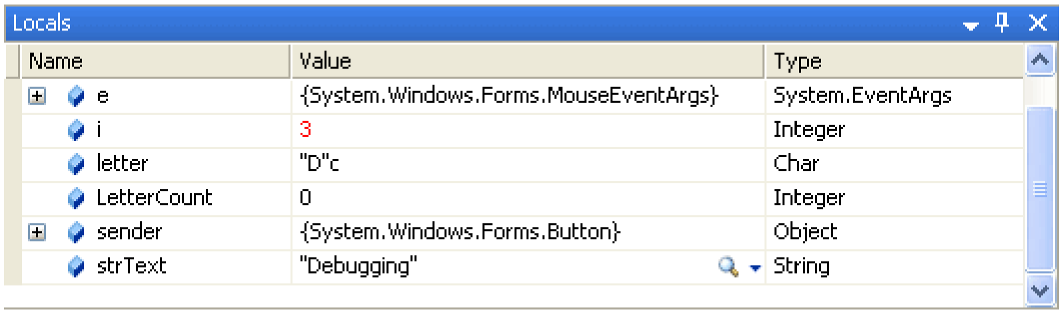
While your programme is still in Debug mode (the yellow line will still be there, if it is), click **Debug > Windows > Locals** from the menu bar. You should see the following in the bottom left of your screen:



Locals

Name	Value	Type
e	{System.Windows.Forms.MouseEventArgs}	System.EventArgs
i	1	Integer
letter	Nothing	Char
LetterCount	0	Integer
sender	{System.Windows.Forms.Button}	Object
strText	"Debugging"	String

Locals means "Local variables". That is, variables declared in this section of the code. The variables **i**, **letter** and **LetterCount** are showing in the window. The value of these variables is also displayed: **0**, **Nothing** and **0**. Press F10 and these values will change. This is the Locals window after one go round the loop:



Locals

Name	Value	Type
e	{System.Windows.Forms.MouseEventArgs}	System.EventArgs
i	3	Integer
letter	"D"c	Char
LetterCount	0	Integer
sender	{System.Windows.Forms.Button}	Object
strText	"Debugging"	String

The variable **i** is now **2**; letter is "**D**", and **LetterCount**, is still **0**. Keep pressing F10 and go round the loop a few times. What do you notice?

You should notice that the value in letter never moves on. It is "D" all the time. And that's why LetterCount never gets beyond 0.

Exercise

Why does **LetterCount** never gets beyond 0? Correct the code so that your textbox displays the correct answer of 3 when the programme is run.

And that's it for Error checking and debugging. It can often be an art form in itself. But one well worth persevering with: it could save you a lot of headaches!

In the next section, we'll move on and have a look at what arrays are.

What is an Array?

Computer Tutorials List

Beginners Computing
Microsoft Word
Microsoft Excel
Web Design
Cascading Style Sheets
Javascript
> Visual Basic .NET
Beginners PHP

Qualifications and Certificates

In this section, you are going to learn all about the power of arrays, and how easy they can make your programming life. First, you need to know what an array is.

What is an array?

So far you've been using variables quite a lot. You've put numbers into variables, and you've put text into variables. But you've only done this one at a time: you've put one number into a variable, or one string of text. You've been doing this:

```
Dim MyNumber As Integer

MyNumber = 5
```

Or this

```
Dim MyText As String

MyText = "A String is really just text"
```

Or even this:

```
Dim MyNumber As Integer = 5
```

So one variable was holding one piece of information. An array is a variable that can hold more than one piece of information at a time. The MyNumber variable above held one number 5. If you had an array variable called MyNumbers - plural - you could hold more than one number at a time. You set them up like this:

```
Dim MyNumbers(4) As Integer

MyNumbers(0) = 1
MyNumbers(1) = 2
MyNumbers(2) = 3
MyNumbers(3) = 4
MyNumbers(4) = 5
```

When you set up an array with the **Dim** word, you put the name of your array variable, and tell Visual Basic how many items you want to store in the array. But you need to use parentheses around your figure. You then assign your data to a position in the array. In the example above we've set up an Integer array with 5 items in it. We've then said put number 1 into array position 0, put number 2 into array position 1, put number 3 into array position 2, and so on.

You might be thinking that the array was set to the number 4 - **MyNumbers(4)** - but always remember that an array starts counting at zero, and the first position in your array will be zero.

So that's what an array is - **a variable that can hold more than one piece of data at a time** -but how do they work? A programming example might help to clear things up.

- Start a new VB project.
- Add a **Button** to your Form.
- Set the **Text** property of the Button to "**Integer Array**"
- Put the following code behind your button:

```
Dim MyNumbers(4) As Integer

MyNumbers(0) = 1
MyNumbers(1) = 2
MyNumbers(2) = 3
MyNumbers(3) = 4
MyNumbers(4) = 5

MsgBox("First Number is: " & MyNumbers(0))
MsgBox("Second Number is: " & MyNumbers(1))
MsgBox("Third Number is: " & MyNumbers(2))
MsgBox("Fourth Number is: " & MyNumbers(3))
MsgBox("Fifth Number is: " & MyNumbers(4))
```

Test out the programme when you are finished. The numbers 10 to 50 should have been displayed in your message boxes.

In the code, we first set up an Integer array with 5 items in it.

```
Dim MyNumbers(4) As Integer
```

We then assigned values to each position in the array.

```
MyNumbers(0) = 1
```

To get at the values in the array, and display them in messages boxes, we just used the array name, followed by the position in the array.

```
MsgBox("First Number is: " & MyNumbers(0))
```

So we've said, "Display whatever number is in array position 0, then display whatever number is in array position 1 ... " and so on.

Add another messages box statement on a line below the others. Put this:

MsgBox("Sixth Number is: " & MyNumbers(5))

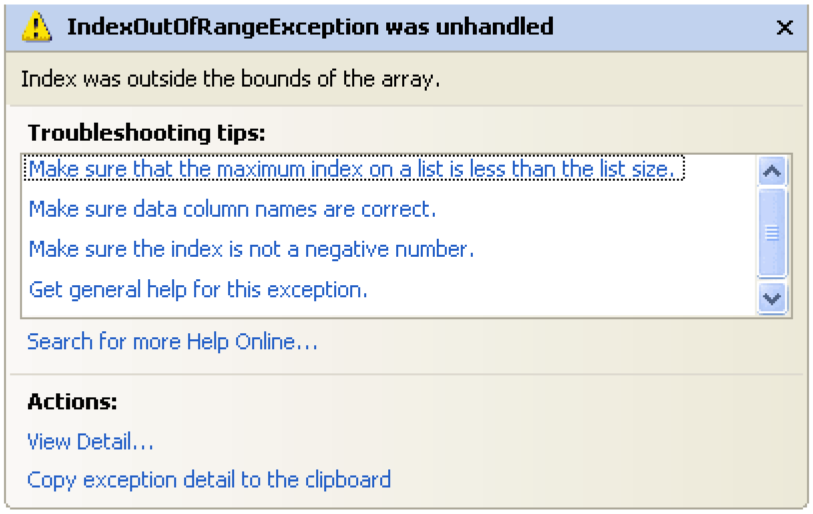
Run your programme again, and click the Button.

What happened? To explain what went wrong, and why, click the next part below.

Arrays and the Index Number

This lesson is part of an ongoing tutorial. This first part is here: What is an Array?

In the previous part, you added some code to a button in order to test out arrays. When you clicked your button, you probably got an error box popping up, telling you that the "**Index was outside the bounds of the array**." This is the error message you may have received:



IndexOutOfRangeException was unhandled

Index was outside the bounds of the array.

Troubleshooting tips:

Make sure that the maximum index on a list is less than the list size.
Make sure data column names are correct.
Make sure the index is not a negative number.
Get general help for this exception.

Search for more Help Online...

Actions:

View Detail...
Copy exception detail to the clipboard

The **Index** the error message is talking about is the figure in parentheses in your array. We set up this array

Dim MyNumbers(4) As Integer

And the highest Index number is therefore **4**. But we tried to display something at index number **5**:

MyNumbers(5)

Visual Basic said "Wait a minute, the idiot hasn't got a position number 5!" So it stopped the programme and gave you an error message. Delete this line from your code:

```
MsgBox("Sixth Number is: " & MyNumbers(5))
```

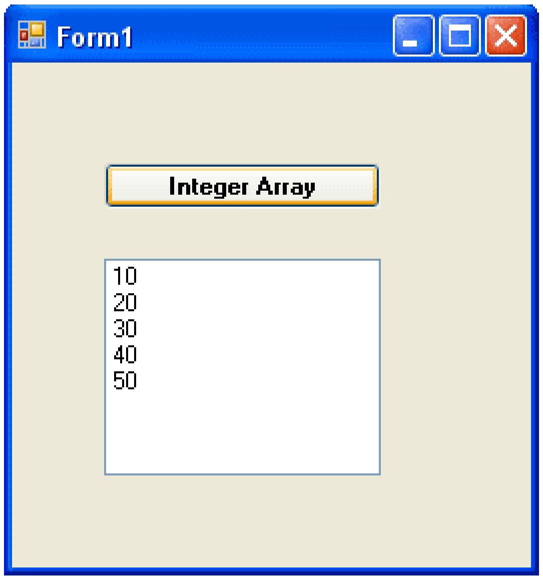
So the way to get at information held in an array is through its Index number - "What's at array position 0? What's at array position 1?" A very handy way to get at the information in your array is by accessing its Index number in a For Loop.

So that you don't have all those message boxes popping up, we can display the results in a List Box.

Add a List Box to your form. Make it fairly wide, and just leave it on the default **Name** of **ListBox1**. Then change your code to the following (the new code is in Bold, red text):

```
Dim MyNumbers(4) As Integer

Dim i As Integer

MyNumbers(0) = 10
MyNumbers(1) = 20
MyNumbers(2) = 30
MyNumbers(3) = 40
MyNumbers(4) = 50

For i = 0 To 4
ListBox1.Items.Add(MyNumbers(i))
Next i
```

Run your programme, and click your button. Your form should look something like this one:



```
10
20
30
40
50
```

The first time round the loop, the variable called **i** will hold the number **0**. Visual Basic will test to see if our end condition is met. The end condition was "**Loop until the variable i holds the number 4**". The variable **i** only holds the number **0**, so Visual Basic drops down to the next line:

ListBox1.Items.Add(MyNumbers(i))

And what is inside the variable **i**? The number 0. So what's really getting adding to the List Box is this:

MyNumbers(0)

In other words, "**Add to the List Box whatever is inside the array at position number 0**"

The next time round the loop, the variable i will hold the number 1. So this gets executed

ListBox1.Items.Add(MyNumbers(1))

And the loop continues round and around, adding whatever is inside our array until the end condition for the loop is met.

Change the first line of the For loop to this:

For i = 0 To 5

Can you guess what will happen? Try it an see. Make sure you know why you get the error message before moving on.

Arrays and Strings of Text

Arrays can hold other types of data, too. They can hold Strings of text.

- Put another Button on your Form
- Set the Text property to "String Array"
- Put the following code behind your Button

```
Dim MyText(4) As String

Dim i As Integer

MyText(0) = "This"
MyText(1) = "is"
MyText(2) = "a"
MyText(3) = "String"
MyText(4) = "Array"

For i = 0 To 4
ListBox1.Items.Add(MyText(i))
Next i
```

When you have finished, run the programme and click your new button. The text you put into the 5 array positions should display in the List Box.

Again, the same process is at work: Set up an array, and specify how many items you want to hold in the array; assign your data to each position; go round a loop and access whatever is in each position of the array.

In the next part, we'll take a closer look at assigning values to an Array.

Assigning Values to an Array

This lesson is part of an ongoing tutorial. This first part is here: What is an Array?

There are a number of way you can put data into each position of an array. The code we just wrote for the two buttons had known values stored into each position. We knew we that we wanted the numbers 1 to 5 to be stored into our Integer array, and we knew that we wanted the text "This is a String Array" stored into our String array.

But you don't have to know what the values are. You can assign values straight from a Textbox into the position of your array. Like this:

```
MyNumbers(0) = Val(Textbox1.Text)
MyNumbers(1) = Val(Textbox2.Text)
etc
```

With that code, whatever you typed into the Textboxes on your Form would be stored into the positions of your array. The same would be true of a String Array:

```
MyNumbers(0) = Textbox1.Text
MyNumbers(1) = Textbox2.Text
etc
```

But do we have to keep typing out a value for each and every position of our array. What if we had an array with a hundred items in it, **MyText(99)**? Would we have to type out text for all one hundred positions of the array?

Well, obviously not. You can use code to assign values to your array. Here is an example where we don't type out values for all positions of an array. It's the times table again. This time we'll use an array. And we'll write a line of code to assign values to each position of the array.

- First, add another **Button** to your form.
- Set the **Text** Property to "**Times Table Array**"
- Add a **Textbox** to your Form
- Set the **Text** Property to a blank string (in other words, delete Textbox1 from the Text property)
- Add a **Label** near the Textbox
- Set the **Text** property of the Label to "**Which Times Table do you want?**"
- Now double click your new button to get at the code window. Add the following code:

```
Dim numbers(10) As Integer
Dim times As Integer
Dim StoreAnswer As Integer
Dim i As Integer

ListBox1.Items.Clear()

times = Val(TextBox1.Text)

For i = 1 To 10

StoreAnswer = i * times
numbers(i) = StoreAnswer
ListBox1.Items.Add(times & " times " & i & " = " & numbers(i))
```

Next i

Run the programme. Enter a number in your new text box, and then click the **Times Table Array** button. The times table for the number should have been printed.

No more reading these lessons online - get the eBook here!

At the top of the code we set up three normal Integer variables, **i, times** and **StoreAnswer**. (We didn't really need the StoreAnswer variable, but it is here to make the code more readable.) We also set up an array. (Notice that we set it to 10. This actually gives us 11 positions in the array. But we're only putting something in positions 1 to 10. This is because it is more convenient for us, and for our Loop.)

Dim numbers(10) As Integer

We need to know what number we are going to be multiplying by, which times table we're working out. We get this number from the Textbox, and can assign it directly to the variable times

times = Val(Textbox1.Text)

We can then set up a For Loop. Inside the For Loop is where we'll assign values to each position of our array:

numbers(i) = StoreAnswer

First time around the loop, the variable i will hold a value of 1. So the second position of our array, **numbers(1)** will be assigned whatever is in the variable **StoreAnswer**

The second time around the loop, the variable **i** will hold a value of 2. So the second position of our array, **numbers(2)**, will again be assigned whatever is in the variable **StoreAnswer**

We go round and round the loop assigning values to all ten positions of our array.

The other two lines of code inside the array just work out the times tables, and **Adds** the answer to the List Box. Study them, and make sure you understand how they work.

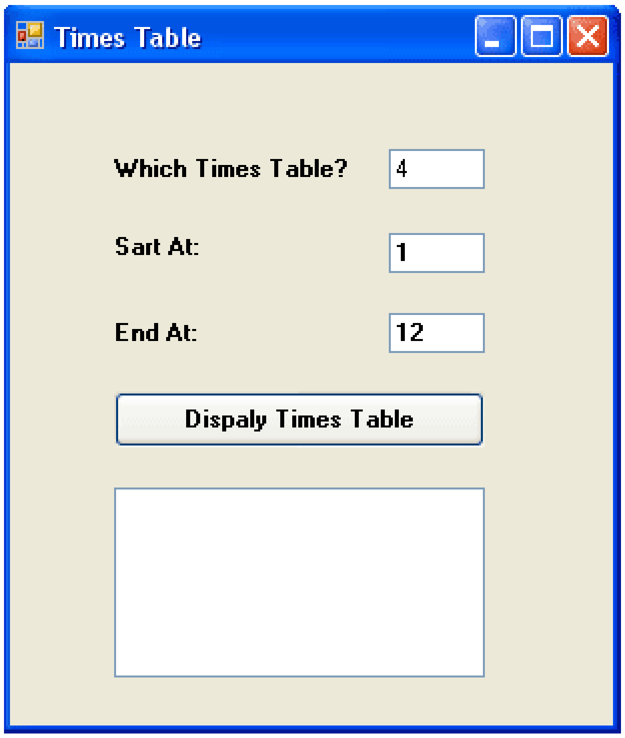
But the point of this is to demonstrate that you can use code to assign a value to a position in an array.

In the next part, we'll take a look at situations where you don't know how many items will be in an array.

Arrays where the Boundaries are not known

This lesson is part of an ongoing tutorial. This first part is here: [What is an Array?]()

Study the following Form:



In the above Form, the user is invited to enter values into three Textboxes. The first Textbox is for whatever times table he or she wants. The second Textbox asks for the starting value of the times table. The third Textbox is for the end number of the times table. In other words, 1 times 4, 2 times 4, 3 times 4, right up to 12 times 4.

The point is that the user can enter any values he or she wants. We won't know until they are entered, and the button is clicked. Up until now, we've used an array with a fixed size. Our previous times table programme only went up to 10, and it started at 1. We used this to set up our array:

Dim numbers(10) As Integer

But that array would be no good for the above Form. Our array only held 11 positions. The user definitely wants the 4 times table right up to 12. Is there any way we can set up an array where the number of positions is not known? How can we set up a Non-Fixed size array?

You do it like this. First set up an array with empty brackets

```
Dim numbers( ) As Integer
```

Next, pass the values from the Text Boxes to some variables

```
times = Val(Textbox1.Text)
startAt = Val(Textbox2.Text)
endAt = Val(Textbox3.Text)
```

We can then use these values to reset the array. You reset an array by using the ReDim word. You then specify the new values. Like this:

```
ReDim numbers(endAt)
```

Our original array did not have its size set - Dim numbers( ) As Integer. So we've got the end number from the Textbox. When we reset an array, we can use this new value. Since our user entered the value 12 for the end number, our array is now really this:

```
Dim numbers(12) As Integer
```

We can use the same variables for our For Loop. Then we can go round and round the loop assigning values to our array.

No more reading these lessons online - get the eBook here!

To test this concept out, either start a new project, or amend the one you have displayed. Create three textboxes and labels. And add a new Button. Double click your button to open the code window. Then add the following code (the important line is in red):

```
Dim numbers() As Integer
Dim startAt As Integer
Dim endAt As Integer
Dim times As Integer
Dim StoreAnswer As Integer
Dim i As Integer
```

```
times = Val(TextBox1.Text)
startAt = Val(TextBox2.Text)
endAt = Val(TextBox3.Text)

ReDim numbers(endAt)

For i = startAt To endAt

StoreAnswer = i * times
numbers(i) = StoreAnswer
ListBox1.Items.Add(times & " times " & i & " = " & numbers(i))

Next i
```

When you're finished, run your programme and test it out. Click the button and the times table should appear in the List Box.

And that is how to set up an array when you don't know what size the array is going to be - set up an array with empty brackets. Reset the array with the ReDim word, and then give it some new values.

Arrays can be a very tricky subject, and they dp take some getting used to. But they are well worth your time and effort - they'll make your coding life a lot easier!

We'll move on to another subject - working with Strings of text.

String Manipulation in VB .NET

Humans are far from perfect. Especially when they are entering data into textboxes! Sometimes they won't enter any details at all in the boxes you want them to. And then when they do enter something, they often get it wrong. Sometimes on purpose, just to trip you up. By manipulating Strings of data, we can check things like text in a textbox to see if it is correct, and thereby gain some control of the user's actions.

First, let's take a closer look at the String variable type.

The String Variable Type

There's more to the string variable type than meets the eye. You've used them a lot to store text. But the string variable types come with a lot of inbuilt power. Strings have their own properties and methods, just like a textbox or label or form does. That's because strings are objects. (In fact, all variables are objects in VB.NET, including the number variables.) In a later section, we'll be going into a bit more detail on objects. For now, think of them as things that can be manipulate - Like the textbox, label and form just mentioned.

And strings variables can be directly manipulated, too. An example should clear things up.

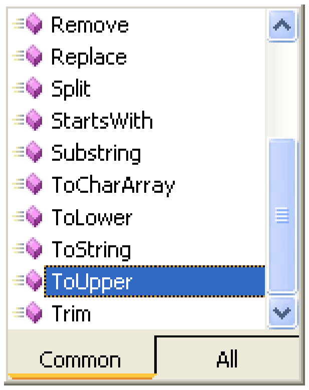
- Start a new project.
- Add two textboxes and a button to your new form.
- For Textbox1, set the Text property to "**string variables**".
- Double click the button to open the coding window.
- Type the following as the code for the button:

```
Dim strUpper As String

strUpper = TextBox1.Text
TextBox2.Text = strUpper.ToUpper( )
```

Run your code and see what happens when you click the button.

You should have found that the text from Textbox1 gets converted to uppercase letters.

The reason it gets converted is because we used the **ToUpper** method of the string variable. When you typed the full stop after the variable name, you probably saw this pop up box:



Simply double click the method you want, and it's added to your code.

strUpper.ToUpper( )

Notice that the name of the variable you want to do something with comes first. Then, after the full stop, you add the name of the method.

It's easy to guess what some of the methods do (like ToLower), but others are a bit more enigmatic (Like Substring).

In this section, we'll go through some of the string methods to see what they do, and how useful they can be in your code.

Before we start, here's a full list of the methods that a string variable can access (it's a bit long, so it gets its own window!):

The full list of String methods and properties (9K - needs javascript enabled)

Length and **Chars** on that list above are properties, and not methods. We'll be using these two, and they come in quite useful.

Manipulating data from a Text Box

You already know how to pass data from a Textbox into a variable. Just do this

```
Dim FirstName As String

FirstName = txtFirst.Text
```

Then whatever was in the Textbox you called txtFirst will get transferred directly to the String variable you set up. Once the data is in the variable, you can test it to see if it's the type of data you want. After all, the user could have entered numbers to try and trip you up. Or they could have left it blank.

- Add a new textbox to your form
- Change the **Name** property to txtFirst
- Add a second button to your form
- Set the **Text** property to whatever you want
- Double click the button and add the following code:

```
Dim FirstName As String

FirstName = txtFirst.Text

If FirstName = "" Then

MsgBox "Please enter your First Name in the text box"
Exit Sub

End If
```

In this code, we are passing whatever is in the text box directly to the variable called **FirstName**. We are then testing what is inside the variable with an If statement. We want to test if the user has actually entered something into the text box. If they've left it blank, we want to tell them to try again. We also want to exit the Subroutine. After all, if they got it wrong, we don't want to proceed with any code we might have written.

The way we are testing to see if the user has left the text box blank is this:

```
If FirstName = "" Then
```

We've put two sets of double quotes together. This signifies a string of text that is blank. If the user didn't enter anything at all, then our variable **FirstName** will contain a blank string. This is what we're testing for.

Run the programme and try it out. Don't type anything at all in the textbox, but just click the button. The message box should display.

Now, click inside the textbox. Hit the space bar three times. And then click the button. Did the Message box display?

So why didn't it? After all, there was nothing in the textbox. Isn't that an blank string? What was passed to the variable?

Well, when you press the space bar Visual Basic counts that as a text character. So when you press the space bar three times what is in the variable is this:

```
FirstName = " "
```

and not this:

```
FirstName = ""
```

The two are entirely different, according to Visual Basic. After all, you might have wanted three spaces!

So how can we check to see if there is anything at all in our textbox? How do we defeat the user who has tried to fool us by hitting the space bar a number of times?

In the rest if this section, you'll learn about the various ways you can use the String methods and properties to examine what a particular string contains. First up is the Trim method.

The Trim Method in VB .NET

This lesson is part of an ongoing tutorial. The first part is here: String manipulation

In the previous part you added code to transfer some text that was in a Textbox over to a variable. The code you wrote was this:

```
Dim FirstName As String

FirstName = txtFirst.Text

If FirstName = "" Then

MsgBox "Please enter your First Name in the text box"
Exit Sub

End If
```

But the user could press the spacebar in your textbox. Although there would be no letters or numbers inthe textbox, the above error checking wouldn't work - all of the blank spaces would get passed to your variable. We can use a String method called Trim to solve this problem.

The Trim Method

One of the methods on our list was **Trim**. What this does is to trim any leading or trailing blank spaces from a string. So if the string was **" Text"**, then Trim would delete those spaces for you, leaving just **"Text"**.

You use it in your code. Like this:

```
FirstName = txtFirst.Text
FirstName = FirstName.Trim
```

First, we put the text from the textbox into a variable called **FirstName**. Then we said "assign to the variable **FirstName** (FirstName =) the value of **FirstName** trimmed (FirstName.Trim)".

Again, though, we're just adding the method we want after the variable name. VB will take care of the trimming for us.

Another way to Trim is to use the Trim() function directly. Like this:

```
FirstName = Trim(txtFirst.Text)
```

What you are trimming here is any blank spaces from around text entered directly in the text box called txtFirst

But we now have a way to thwart that user who is trying to trip us up by entering blank spaces into our text box. If you were programming a Form where the First Name was going into a database, then it's essential you trap anything like this.

OK, we've tested to see if the First Name text box was empty. Is there anything else we can do? What if our clever-clogs user tries to fool us again. This time he (they're always "he's"!) decides to enter some numbers. He claims his name is "**George12345**". Is there anything we can do to stop his little games? Is there a way to test that the data entered into a text box was text and not numbers?

Indeed there is! We'll use the **Chars** Property to see if we can check for numbers and text. You'll also see the difference between **Chars** and **Char**.

Char and Chars in VB .NET

Char (NOT Chars)

Char is a variable type. It can hold one character at a time (the Char is short for Character). You set it up like this:

```
Dim OneCharacter As Char
```

You can then store a character in the variable like this:

```
OneCharacter = "A"
```

Or like this:

```
Dim OneCharacter As Char = "a"
```

You can even do this:

```
Dim OneCharacter As Char = "apple"
```

But if you try to put a whole word into a Char variable, only the first letter will be retained.

So what good is the Char variable type?

Well, a common use for it is to transfer one letter at a time from a string, and then test to see what this character is. You can test to see if it's a number, for example. Or perhaps to test if the string contains an "@" symbol for a valid email address. We'll test for a number. In the process, we can study the Length property of string variables.

Add another textbox and a button to your form from the first part of the tutorial. Change the **Name** property of the textbox to **txtChars**. For the **Text** property of the Textbox, enter "**George123**". Double click the new button and enter the following variable declarations:

```
Dim OneCharacter As Char
Dim FirstName As String
Dim i As Integer
Dim TextLength As Integer
```

Remember what we're going to be doing here. We're going to put the text from the textbox into a string variable. Then we'll loop round every character in the string to see if it's a number.

So the next line to add to your code is the one that transfers the text from the textbox to the string variable. Add this:

```
FirstName = Trim(txtChars.Text)
```

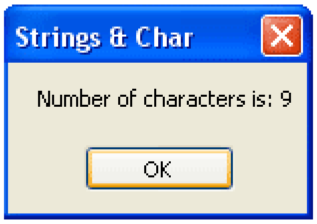
The next thing we need is the length of the string. We need this for the end value of our loop. The length property will give us this answer. So add this line:

```
TextLength = FirstName.Length
```

The length property of a string variable tells you how many characters are in the string. You can add a message box to test out your code so far:

```
MsgBox("Number of characters is: " & TextLength)
```

Run your programme. Click the button and test out your code. You should see a message box popping up like this one:



Number of characters is: 9

So we've found out that "**George123**" has 9 characters.

We can now loop round each character in the string and test which ones are the numbers. Add the following For loop to your code (you can delete or comment out your message box line now):

```
For i = 0 To TextLength - 1

Next i
```

So the For loop starts at zero and ends at the length of the text, minus 1. (It will loop from 0 to 8 in our code - 9 characters. We'll see why you have to deduct 1 soon.

Inside of our loop, we need to grab one character at a time, and then put it into our **Char** variable. You can do that with the **Chars()** Property of the string variable type.

Chars (NOT Char)

Chars is a method of the String variable type. You can use it on any length of string, not just a Char variable. And that's the difference between the two: Char is a variable type, while **Chars** is a method you can use on Strings.

Chars works like this:

```
OneCharacter = FirstName.Chars(i)
```

You type the name of your variable, then after the full stop you add **Chars()**. Inside of the round brackets, you need a number. This number is the position in the string you want to grab. So if you wanted the third letter of a string variable, you'd put this:

```
Dim SomeString As String
Dim OneCharacter As Char

SomeString = "George123"
OneCharacter = SomeString.Chars(2)
```

The variable **OneCharacter** would then hold the third letter - "**o**".

The reason we've put 2 inside of the round brackets and not 3 is because VB starts counting the characters from zero, and NOT 1. And that's why our For Loop is this:

```
For i = 0 To TextLength - 1
```

You have to deduct 1 because the Chars() count starts at zero.

So amend your For Loop to this:

```
For i = 0 To TextLength - 1
```

```
OneCharacter = FirstName.Chars(i)
MsgBox(OneCharacter)

Next i
```

Run your code, and then click your button. You should get a message box displaying. In fact, you'll get 9 message boxes, one for each character in the string!

Ok, try these exercises to test your new knowledge.

Exercise

Add an If statement to your For Loop. Check each character of the string "**George123**". If you find a number, display a suitable message, something like "A number was found". Exit the for loop when the first number is found.

To check if a variable is a number, you can use the IsNumeric( ) function. This function will return either True or False. In other words, if the variable being checked is a number, then IsNumeric( ) is True; if IsNumeric( ) is not a number then False gets returned.

```
If IsNumeric(OneCharacter) Then
```

Exercise

Amend your code to keep a count of how many characters in the string are numbers. Display the count in a message box.

In the next part, we'll take a look at another useful String method - InStr.

The InStr() Method in VB .NET

String Position

The **InStr( )** method of string variables tells you what the position of one string is inside another. For example, if your string was "**me@me.com**" and you wanted to know if the string contained the @ symbol, you could use InStr( ) Method. You would use it like this

```
FirstString = "me@me.com"
SecondString = "@"

position = InStr(FirstString, SecondString)
```

The variable **FirstString** is the string we want to search; **SecondString** is what we want to search for. You can specify a starting position for the search to begin. If you do, this number goes at the start (the default is zero):

```
position = InStr(1, FirstString, SecondString)
```

The variable called **position** has to be an integer variable. That's because the **InStr()** Method returns a number, and not text. In the code above, **position** would have a value of **3**. That's because the @ symbols starts at the third letter of "me@me.com".

(Note: the InStr() Method starts counting at 1, and not zero like **Chars()**, which is very confusing!)

If the string you're searching for is not found, then the value placed inside of your integer variable (**position** in our case) is zero. That enables you to code something like this:

```
If position = 0 Then

MsgBox "Not a Valid email address: There was No @ Sign"

End If
```

Another useful string method is Substring. We'll see how to use it in the next part.

The Substring() Method in VB .NET

Substring

Another useful string method is **Substring**. This allows you to grab one string within another. (For example, if you wanted to grab the "**.com**" from the email address "**me@me.com**")

In between the round brackets of **Substring( )**, you specify a starting position, and then how many characters you want to grab (the count starts at zero again). Like this:

```
Dim Email as String
Dim DotCom as String

Email = "me@me.com"
DotCom = Email.Substring(5, 4)

MsgBox(DotCom)
```

The message box would then display the characters grabbed from the string, in this case the "**.com**" at the end (start at position **5** in the string and grab **4** characters).

You could also do a check to see if an email address ended in ".com" like this:

```
Dim Email As String
Dim DotCom As String

Email = "me@me.con"
DotCom = Email.Substring(Email.Length - 4, 4)

If DotCom = ".com" Then
MsgBox("Ends in Dot Com")
Else
MsgBox("Doesn't End in Dot Com")
End If
```

The starting position for Substring( ) this time is "**Email.Length - 4**". This is the length of the string variable called **Email**, minus 4 characters. The other 4 means "grab four characters"

You have to be careful, though. If there wasn't four characters to grab, VB would give you an error message.

We could replace the Chars() For loop code we wrote earlier with a Substring() method. The result would be the same. Here's the code:

```
For i = 0 To TextLength - 1
OneCharacter = FirstName.Substring(i, 1)
MsgBox OneCharacter
Next i
```

So we're saying, "Start grabbing characters from the position **i**. Just grab one character".

Substring and Chars are very useful methods to use when you want to check the letters in a string of text.

In the next part, we'll take a look at the Equals, Replace, and Insert Methods.

Equals, Replace, and Insert Methods

The Equals Method

In code previously, we had this:

```
If DotCom = ".com" Then
MsgBox("Ends in Dot Com")
Else
MsgBox("Doesn't End in Dot Com")
End If
```

You can use the **Equals** method of string variables in the first line, instead of an equals (=) sign:

```
If DotCom.Equals(".com") Then
```

So after the name of your string variable comes the full stop. Then select "Equals" from the popup list. In between the round brackets, you type the string (or variable name) that you want VB to compare.

The Equals method is used to compare one string of text against another. If they're the same a value of True is returned, else it's False.

No more reading these lessons online - get the eBook here!

The Replace Method

You can replace text in one string with some other text. The process is fairly straightforward. Here's some code that uses replace. Add a button to a form and test it out:

```
Dim OldText As String
Dim NewText As String

OldText = "This is some test"
NewText = OldText.Replace("test", "text")

MsgBox(OldText)
MsgBox(NewText)
```

When you run the programme, the first message box will say "This is some test" and the second box will say "This is some text".

The text you want to get rid of comes first. Then after a comma, type the new text. You can use string variables rather than direct text surrounded by double quotes, for example:

```
Dim NewWord As String = "Text"
NewText = OldText.Replace("test", NewWord)
```

The Insert Method

You can also insert some new text into an string. Here's some code to try out:

```
Dim SomeText As String
Dim NewText As String

SomeText = "This some text"
NewText = SomeText.Insert(5, "is ")

MsgBox(SomeText)
MsgBox(NewText)
```

The 5 in round brackets means start at position 5 in the string variable SomeText (the count starts at zero). You then type the text that you want inserted. You can use a variable name instead of direct text surrounded by quotes.

In the next part, we'll take a look at how to use Split() and Join in VB .NET.

Split() and Join() in VB .NET

Two very useful string variable methods are **Split** and **Join**. **Split()** allows you to split a line of text and put each element (word or phrase) into an array; **Join()** allows you to join elements of an array into one line of text. An example or two might clear this up.

The Split() Method

In a later project, you'll have to open up text file and read it's contents. You can read the text file line by line, and each line might be something like this:

"UserName1, Password1, UserName2, Password2, UserName3, Password3"

The programming problem is to separate each word. You can use Split for this. Each word would then be separated, ready for you to place into an array.

Here's an example for you to try out. (It's better to put this code behind a new button):

```
Dim LineOfText As String
Dim i As Integer
Dim aryTextFile() As String

LineOfText = "UserName1, Password1, UserName2, Password2"

aryTextFile = LineOfText.Split(",")

For i = 0 To UBound(aryTextFile)
MsgBox(aryTextFile(i))
Next i
```

Notice the line that sets up an array:

```
Dim aryTextFile() As String
```

We don't know how many elements will be in the array (how many words on each line), so we leave the round brackets blank.

The next line just put some text into a variable called LineOfText. But this can come from a text file that you open with code.

The line that does the splitting comes next:

```
aryTextFile = LineOfText.Split(",")
```

Notice that aryTextFile has now lost its round brackets. If you put them in, you get an error. The use of the Split() method, though, is this:

```
LineOfText.Split(",")
```

After the name of your variable and the full stop, select (or type) the word Split. In between round brackets you put what is know as the separator. This is the symbol or punctuation mark that you are using to separate each element of your line. In our case, we're using a comma to separate the words in the line. But you can use anything you like (a hyphen, for example).

When VB finishes the splitting, it fills up your array. Each element will occupy one slot in your array. So in our example, **aryTextFile(0)** will hold a value of **UserName1**, **aryTextFile(1)** will hold a value of **Password1**, etc.

The For loop is there to show you how to loop round each element in your array, and displays the results in a message box:

```
For i = 0 To UBound(aryTextFile)
MsgBox(aryTextFile(i))
Next i
```

The first line includes this:

UBound(aryTextFile)

UBound means Upper Boundary of an array. In between the round brackets of UBound() you type the name of your array. Notice, though that the round brackets of the array have gone missing again.

So if your array was this:

MyArray(9)

The Upper Boundary of the array would be 9. So the end of the For Loop would then be 9. You code like this when you don't know how many elements are in your array.

The message box just displays what is in each position of your array:

MsgBox(aryTextFile(i))

The point is that all of the words from your line of text have been split and placed into an array. You now need to know how to put the text back together again.

The Join() Method

The Join method is used when you want to join the elements of an array back together again. Here's some code which does exactly that.

```
Dim LineOfText As String
Dim i As Integer
Dim aryTextFile(3) As String

aryTextFile(0) = "UserName1"
aryTextFile(1) = "Password1"
aryTextFile(2) = "UserName2"
aryTextFile(3) = "Password2"

LineOfText = String.Join("-", aryTextFile)

MsgBox(LineOfText)
```

The line that joins each element in the array is this:

```
LineOfText = String.Join("-", aryTextFile)
```

(NOTE: If you have an older version of the VB NET software, use **LineOfText**.Join instead of **String**.Join.)

In between the round brackets of Join(), you first type what you want to use as a separator. Here, we're using an hyphen as a separator. Next, you put the name of your array. Again the round brackets from the array have gone missing.

When the line executes, the variable LineOfText will hold the following:

"UserName1-Password1-UserName2-Password2"

Once you have the array elements joined together, you could then write the line back to your text file.

Split and Join can be very useful indeed. Especially when you're working with text files. Speaking of which, the next section deals with exactly this subject.

Text Files and VB .NET

There is a very useful object in VB.NET called **System.IO** (the IO stands for **Input** and **Output**). You can use this object to read and write to text files.

We're going to be having a closer look at objects (and what System is) in the next section. For now, let's just see how to open up a text file using the System.IO object.

First, here's an explanation of just what we mean by "text file".

What is a Text File?

The files on your computer all end in a three letter extensions. Microsoft Word files will have a different three letter extension from Microsoft Excel files. The extension is used to identify one file type from another. That way, Excel won't try to open Word files, or vice versa. You can just write some code to strip the last three letters from the file name, and then check that these three letters are the ones you want. Rather like the code you wrote to strip the last three letters from an email address.

Text files have an extension that ends in **.txt**. The Windows operating system gives you a good, basic Text Editor in Notepad. The Notepad programme allows you to save files with the **.txt** extension. In other words, as Text Files. These Text Files can then be opened by a wide variety of programmes.

A simple text file like this is called a Sequential File, and that is what we will be opening here. So let's begin.

How to Open a Text File in VB .NET

The ability to open up a text file and read its contents can be very useful to you in your programming life. You might have a text file containing quiz questions and answers, for example. You could read the questions and answers from a text file and create your own "Who wants to be a Millionaire" game. Or you might want to save some data associated with your programme, and then open it up again when the programme starts. Well see how to open up a text file in VB .NET right now. In a later section, you'll learn how to save data to a text file.

To open up a text file, you need to create something called a "**StreamReader**". This, as its name suggests, reads streams of text. The **StreamReader** is an object available to **System.IO**. You create a StreamReader like this:

Dim FILE_NAME As String = "C:\test.txt"

Dim objReader As New System.IO.StreamReader(FILE_NAME)

The first line just sets up a string variable called FILE_NAME. We store the path and name of our text file inside of the string variable:

= "C:\test.txt"

We're saying that there is a text file called test which is at the location (path) "C:\".

You set up the **StreamReader** to be a variable, just like a String or Integer variable. But we're setting up this variable differently:

Dim objReader As New System.IO.StreamReader(FILE_NAME)

We've called the variable **objReader**. Then, after the "**As**" word comes "**New**". This means "Create a New Object". The type of object we want to create is a **StreamReader** object:

System.IO.StreamReader

Sysytem is the main object. **IO** is an object within System. And **StreamReader** is an object within IO.

StreamReader needs the name of a file to Read. This goes between a pair of round brackets:

System.IO.StreamReader(FILE_NAME)

VB will then assign all of this to the variable called **objReader**. So instead of assigning say 10 to an Integer variable, you are assigning a StreamReader to a variable.

No more reading these lessons online - get the eBook here!

Read To End

But this won't do you any good. We haven't actually opened the text file yet. We've just told VB where the text file is and what object to open it with. You do the opening like this:

TextBox1.Text = objReader.ReadToEnd

Now that objReader is an object variable, it has its own properties and methods available for use (in the same way that the textbox has a Text property).

One of the Methods available to our new StreamReader variable is the ReadToEnd method. This will read the whole of your text, right to the end. We're then popping this in a textbox.

Let's test all this theory out. Do the following:

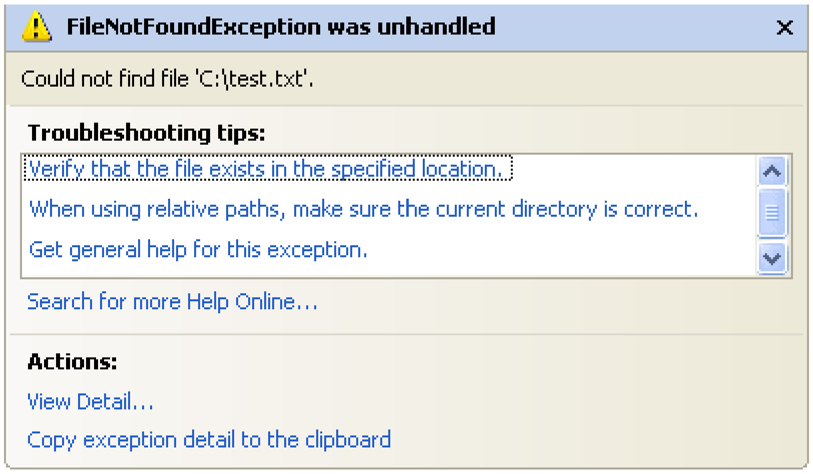
- Start a new project
- Add a textbox to your new form, and just leave it on the default Name of Textbox1
- Set its MultiLine property to True
- Add a Button to your form
- Double click the button and add the following code for it:

```
Dim FILE_NAME As String = "C:\test.txt"

Dim objReader As New System.IO.StreamReader(FILE_NAME)

TextBox1.Text = objReader.ReadToEnd

objReader.Close()
```

The last line closes the StreamReader we set up. You have to close your stream objects after you've used them, otherwise you'll get errors messages.

When you're done, run your programme and click your Button.

Unless you already have a file called **test.txt** at the location **C:\** you'll get this error message popping up:



FileNotFoundException was unhandled

Could not find file 'C:\test.txt'.

Troubleshooting tips:

Verify that the file exists in the specified location.

When using relative paths, make sure the current directory is correct.

Get general help for this exception.

Search for more Help Online...

Actions:

View Detail...

Copy exception detail to the clipboard

The last line spells it out clearly: Could not find file "C:\test.txt". So we were trying to read a text file that doesn't exist.

No more reading these lessons online - get the eBook here!

Does the File Exist?

You can, though, test to see if the file exists. If it does, you can open it; if not, you can display an error message. Amend your code to this (the new lines are in bold, red):

```
Dim FILE_NAME As String = "C:\test.txt"

If System.IO.File.Exists(FILE_NAME) = True Then
Dim objReader As New System.IO.StreamReader(FILE_NAME)
TextBox1.Text = objReader.ReadToEnd
objReader.Close()
Else
```

```
MsgBox("File Does Not Exist")
End If
```

We've now wrapped up our code in an If Statement. The first line of the If Statement is this:

```
If System.IO.File.Exists(FILE_NAME) = True Then
```

This tests to see whether or not a file exists. Again, you start with **System.IO**. Then you access another object of System.IO - the **File** object. This has a method called **Exists**. In between the round brackets, you type the name (or variable) of the file you want to check. The value returned will either be True (if it does exists), or False (if it doesn't).

If the file exist then we can go ahead and create our StreamReader; If it doesn't, we can display a error message for the user.

So that your programme will work, there is a file below called "test.txt". Download this to your computer, in the main **C:\** folder (Right click the file and select Save Target As (IE), or Save Links As (Firefox):

Download the "test.txt" Text File here

When you have done that, run your programme again. Click the button once more, and you should see the text from your file appear in the textbox. (If you get the error message again, it means you haven't copied the file to the right place.)

In the next part, we'll see how to read the contents line by line, instead of all in one go.

Reading a Text File Line by Line

This lesson is part of an ongoing tutorial. The first part is here: How to open a Text File in VB .NET

Quite often, you don't want to read the whole file at once. You want to read it line by line. In which case, instead of using the **ReadToEnd** method, as we did in the previous section, you can use the **ReadLine** method:

The ReadLine method, as its name suggests, reads text one line at a time. In order to do this, though, you need to use a loop. You can then loop round each line and read it into a variable. Here's a coding example:

```
Dim TextLine As String
```

```
Do While objReader.Peek() <> -1
TextLine = TextLine & objReader.ReadLine() & vbNewLine
Loop
```

The first line of the **Do While** loop is rather curious:

Do While objReader.Peek() <> -1

The **Peek** method takes a peek at the incoming text characters. It's looking ahead one character at a time. If it doesn't see any more characters, it will return a value of minus 1. This will signify the end of the text file. Our loop checks for this minus 1, and bails out when **Peek** has this value.

Inside the loop, we're reading each line from the text file and putting into new variable. (We're also adding a new line character on the end. Delete the **& vbNewLine** and see what happens).

objReader.ReadLine()

So the **ReadLine** method reads each line for you, instead of the ReadToEnd method which gets the whole of the text file.

Once you have a line of text in your variable, though, it's up to you to parse it. For example, suppose the line of text coming in from the text file was this:

"UserName1, Password1, UserName2, Password2"

You would then have to chop the line down and do something which each segment. VB won't do this for you! (But you saw how to do this in the last section, when you used things like Split and Substring.)

But what you are doing in the Do Loop is building up your variable with lines of text that are pulled from your text file. Once you have pulled all the text from your file, you can then put it into the text box. Here's our programme once more, with the new code highlighted in bold, red text:

```
Dim FILE_NAME As String = "C:\test.txt"
Dim TextLine As String

If System.IO.File.Exists(FILE_NAME) = True Then

Dim objReader As New System.IO.StreamReader(FILE_NAME)

Do While objReader.Peek() <> -1
TextLine = TextLine & objReader.ReadLine() & vbNewLine
Loop

Textbox1.Text = TextLine
```

```
Else

MsgBox("File Does Not Exist")

End If
```

So inside the loop, we go round building up the **TextLine** variable. Once all the file has been read (when Peek() has a value of -1), we then place it into **Textbox1**.

In the next part, you'll learn how to write to a text file.

How to Write to a Text File in VB .NET

This lesson is part of an ongoing tutorial. The first part is here: How to open a Text File in VB .NET

Writing to a text file is similar to reading a text file. Again we use System.IO. This time, instead of using the StreamReader we use the **StreamWriter**. The StreamWriter is used to write a stream of text to a file.

Add another Button to the form you've been working on. Set the **Text** property of the button to "**Write to File**". Double click your new button to open up the coding window. Add the following:

```
Dim FILE_NAME As String = "C:\test2.txt"

If System.IO.File.Exists(FILE_NAME) = True Then
Dim objWriter As New System.IO.StreamWriter(FILE_NAME)
objWriter.Write(TextBox1.Text)
objWriter.Close()
MsgBox("Text written to file")
Else
MsgBox("File Does Not Exist")
End If
```

Run your programme, and then click your new button.

Unless you have a file called "**test2.txt**", you should see the message box display: "File Does Not Exist."

Once again, VB insists that the file must exist before it can actually do something with it. Which is not unreasonable!

Stop your programme and change this line:

Dim FILE_NAME As String = "C:\test2.txt"

To this:

Dim FILE_NAME As String = "C:\test.txt"

In other words, just change the file name back to test.txt. (Hopefully, you haven't deleted the **test.txt** file from your hard drive!)

Run your programme again. Type something into the textbox, and then click your button. You should see the message box "Text written to file" appear.

But notice that if you open up the text file itself, any text you had previously will be gone - it has been overwritten, rather than appended to. (We'll see how to append text to a file shortly.)

Let's have a look at the code we wrote, though.

Once again, we check to see if the File Exists. If it's True that the file exists, then the first line that gets executed is setting up our variable:

Dim objWriter As New System.IO.StreamWriter(FILE_NAME)

It's almost the same as last time. Only two things have changed: we created a new variable name, **objWriter**, and we're now using **StreamWriter** instead of **StreamReader**. Everything else is the same.

To write the text to our file, we've used this:

objWriter.Write(TextBox1.Text)

After the name of our variable (objWriter), we typed a full stop. The drop down box appeared showing available properties and methods. The "**Write**" method was chosen from the list. In between round brackets, you put what it is you want VB to write to your text file. In our case, this was the text in **Textbox1**. You can also do this:

objWriter.Write("Your Text Here")

The above uses direct text surrounded by double quotes. This is also acceptable:

Dim TheText As String = "Your Text Here"

```
objWriter.Write(TheText)
```

This time, we've put the text inside of a variable. The name of the variable is then typed inside of the round brackets of "Write".

But you don't have to write the whole text at once. You can write line by line. In which case, select **WriteLine** from the available properties and methods. Here's an example of how to use WriteLine:

```
Dim FILE_NAME As String = "C:\test.txt"
Dim i As Integer
Dim aryText(4) As String

aryText(0) = "Mary WriteLine"
aryText(1) = "Had"
aryText(2) = "A"
aryText(3) = "Little"
aryText(4) = "One"

Dim objWriter As New System.IO.StreamWriter(FILE_NAME)

For i = 0 To 4
objWriter.WriteLine(aryText(i))
Next

objWriter.Close()
```

The error checking code has been left out here. But notice the new way to write text to the file:

```
objWriter.WriteLine(aryText(i))
```

We're looping round and writing the contents of an array. Each line of text from the array gets written to our text file. But each line is appended. That is, the text file doesn't get erased after each line has been written. All the lines from the array will be written to the text file. However, if you were to run the code a second time then the contents of the file are erased before the new **WriteLine**() springs into action. In other words, you'd only get one version of "Mary WriteLine had a little one" instead of two.

In the next part we'll see how to add text to a file that already contains text - called appending to a file.

Appending Text to a File in VB .NET

This lesson is part of an ongoing tutorial. The first part is here: How to open a Text File in VB .NET

There will be times when you won't want to erase all the text from your file. You'll only want to add text to what you currently have. In which case you need to **Append**.

Appending text to your file is quite easy.

When you set up the object variable for the StreamWriter, which you did here, you just typed the name and path of the file:

```
Dim objWriter As New System.IO.StreamWriter(FILE_NAME)
```

To append text to a file, you type a comma after your file name then type the word True:

```
Dim objWriter As New System.IO.StreamWriter(FILE_NAME, True)
```

If you want to add some text to the file, you need that True value. If you leave out the True or False, a new file is not created.

Here some code that appends text to the file we wrote to:

```
Dim FILE_NAME As String = "C:\test.txt"
Dim i As Integer
Dim aryText(4) As String

aryText(0) = "Mary WriteLine"
aryText(1) = "Had"
aryText(2) = "Another"
aryText(3) = "Little"
aryText(4) = "One"

Dim objWriter As New System.IO.StreamWriter(FILE_NAME, True)

For i = 0 To 4
objWriter.WriteLine(aryText(i))
Next

objWriter.Close()
MsgBox("Text Appended to the File")
```

The lines that have changed are in bold, red. But as you can see, not much has changed! But try both version and see how they work.

Creating a file if it doesn't exist

If you want to create a file if one doesn't exist, the process is again quite simple:

```
Dim objWriter As New System.IO.StreamWriter(FILE_NAME, False)
```

This time, we've just added the word "False" to the end of FILE_NAME. This will ensure that a new text file is created if one doesn't exist.

In the next part, we'll see how to copy a file in VB .NET.

How to Copy a File in VB .NET

This lesson is part of an ongoing tutorial. The first part is here: How to open a Text File in VB .NET

You can also copy a file that you've created. This time, we don't need the StreamWriter or StreamReader of System.IO. We need the **File** object:

```
System.IO.File
```

This just means "System.IO has an object called File. Use this File object".

File has it's own properties and methods you can use. One of these is **Copy**. Here's some code that makes a copy of our test file .

```
Dim FileToCopy As String
Dim NewCopy As String

FileToCopy = "C:\test.txt"
NewCopy = "C:\NewTest.txt"

If System.IO.File.Exists(FileToCopy) = True Then
System.IO.File.Copy(FileToCopy, NewCopy)
MsgBox("File Copied")
End If
```

The file we want to copy is called "**C:\test.txt**". We've put this inside of a string variable called **FileToCopy**. The name of the new file we want to create, and its location, are assigned to a variable called **NewCopy**.

Next, we have to check to see if the file we're trying to copy exists. Only if it does should we go ahead and copy it. You've met this code before. Inside of the If Statement, we have this:

System.IO.File.Copy(FileToCopy, NewCopy)

We use the **Copy** method of **System.IO.File**. In between the round brackets, you first type the name of the file you want to copy. After a comma, you then type the name of the new file and its new location.

In the next part, we'll see how to Move a file.

How to Move a File with VB .NET

This lesson is part of an ongoing tutorial. The first part is here: How to open a Text File in VB .NET

You move a file in a similar manner as you did to Copying a File - specify a source file and a new destination for it. This time, we use the **Move** method of **System.IO.File**. Here's some code:

```
FileToMove = "C:\test.txt"
MoveLocation = "C:\ TestFolder\test.txt"

If System.IO.File.Exists(FileToMove) = True Then
System.IO.File.Move(FileToMove, MoveLocation)
MsgBox("File Moved")
End If
```

The above code assumes that you have created a folder on your hard drive called "TestFolder":

MoveLocation = "C:\ TestFolder\test.txt"

The file called **test.txt** will then be moved inside of this new location. You can give it a new name, if you want. In which case, just change the name of the file when you're moving it:

MoveLocation = "C:\ TestFolder\NewName.txt"

Again though, the thing to type in the round brackets of the method is first the Source file, then the Destination.

System.IO.File.Move(FileToMove, MoveLocation)

In the final part of this section, you'll learn how to Delete a File using VB .NET code

How to Delete a File in VB .NET

This lesson is part of an ongoing tutorial. The first part is here: How to open a Text File in VB .NET

Deleting a file is quite simple - but dangerous! So be very careful when you're trying out this code. Make sure the file you're going to delete is not needed - you won't be able to restore it from the recycle bin!

To delete a file from your computer, you use the **Delete** method of **System.IO**. Here's some new code for you to try:

```
Dim FileToDelete As String

FileToDelete = "C:\testDelete.txt"

If System.IO.File.Exists(FileToDelete) = True Then

System.IO.File.Delete(FileToDelete)
MsgBox("File Deleted")

End If
```

First, we've set up a string variable called **FileToDelete**. We've then assigned the name of a file to this variable - "C:\testDelete.txt". (We created this file first, and made sure that it was safe to junk it!)

Next, we test to see if the File Exists. In the IF Statement, we then had this:

```
System.IO.File.Delete(FileToDelete)
```

After selecting the **Delete** method, you type the name of the file you want to get rid of. This goes between a pair of round brackets.

And that's it! That's all you need to do to delete a file from your computer, never to see it again. So be very careful when you test out the code!

OK, enough of Text Files. In the next section, we'll look at Functions and Subs.

An Introduction to Functions and Subs

So far, the code you have been writing in these tutorials has mostly been lumped together under one button. The problem with this approach is that your code can get quite long and complex, making it difficult to read, and difficult to put right if something goes wrong. Another approach is to separate some of this code into its own routine. This is where functions and subs come in.

Segements of Code to do a Particular Job

The two terms refer to segments of code that are separate from your main code. You've already met some string functions - **Equals**() and **Substring**(), for example. These functions are built into Visual Basic. But if you could see under the hood, what you'd find is some code to do these jobs for you. All you have to do is to specify, for the Substring() function, what the string is, where you want to start, and how many characters you want to grab. You don't have to worry about how Visual Basic gets your answer, because the code is all wrapped up in a function, ready for you to use over and over again.

And that's the point about Functions and Subs: it's code that you might want to use over and over again. You don't have to keep writing the same code every time you want a specific job doing. Just write the code once, and then when you want to use it, just let Visual Basic know.

Write a Function or Sub to do the Job

Think about error checking when people are entering data on your Forms. You'd probably have a lot of Text Boxes, and want to check that the user was entering the correct data. You don't want people entering numbers in your **First Name** text box, for instance. To check that the user has entered the correct data, you write some error checking code. Except you might have lots of text boxes on the form. If you want to check all the text boxes, you'd have to write the same "checking" code for each text box. Instead of doing that, you can write your own Function or Sub. You only have to write it once. Then when you want to use it, you just refer to it by name, and the code will get executed. We'll soon write a **Sub** that we can use over and over again.

The difference between Functions and Subs

First, though, in case you are wondering what the difference is between a Function and a Sub, it's this: Functions return a value, and Subs don't.

Substring() is a Function, because you want some sort of answer back, and an answer that you can then use elsewhere. You assign the answer to the **Substring**() function to a variable.

A Sub is some code or job that you want VB to get on with and execute. An example is closing a form down and unloading it with Me.Close(). You don't need to return a value, here; you just want VB to close your form down.

An example or two might help to clear things up. We'll start that on the next page.

Create your Own Subs in VB .NET

In the previous part, you learnt what a Sub is, and that it doesn't return a value, unlike a Function. In this part, you'll learn how to create your own Subs in VB .NET. **Sub**, by the way, is short for Subroutine

Here's what we're going to do. We'll set up a text box on a form. The text box will be a First Name text box, and we'll be checking that the user has actually entered something into it.

So, do this:

- Start a new project, and
- Put a Text Box on your new Form.
- Put a button on the Form, too.
- Double lick your button to see the coding window
- Add the following code for the button:

```
Dim TextBoxData As String

TextBoxData = Trim(TextBox1.Text)

If TextBoxData = "" Then
MsgBox("Please Enter your First Name")
End If
```

Run the programme and test it out. Don't enter anything in your textbox, but just click your button. Your message box should display.

Now, all that code is inside the button. More likely than not, we'd be writing more code for that button. In fact, we could be writing lots of code. The code we write could get a bit long and complex. Do we have to have that error checking code in there? And wouldn't we have to type it out all over again, if we wanted to check another textbox from a different button?

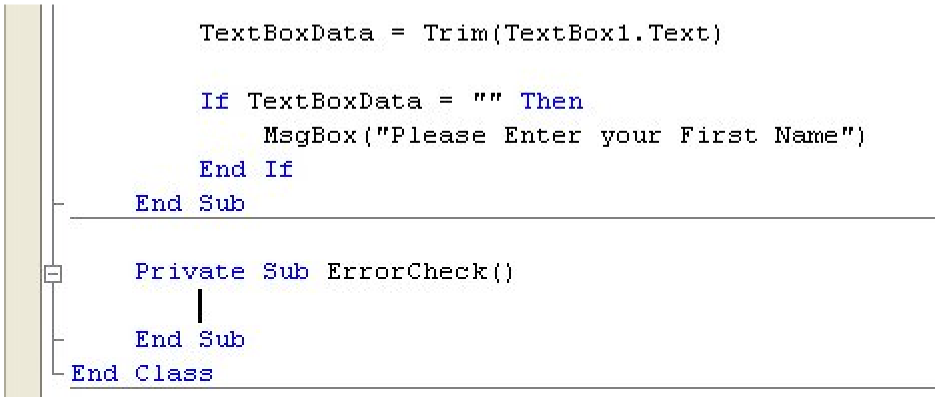
The answer to our two questions are, Not at all, and Yes we would!

To solve the problem, we'll chop that code from the button, and write a Sub for it. To write your own Sub, your cursor needs to be outside of the button code, and on a new line. But before the "End Class". So, on a new line outside the button code type the following:

Private Sub ErrorCheck()

When you've typed that, hit the enter key on your keyboard. Visual Basic will add the **End Sub** for you. The name "**ErrorCheck**" is entirely our own, and we could have called it almost anything we liked. But it's better to stick to something descriptive.

Your code window will then look like this one:



```
        TextBoxData = Trim(TextBox1.Text)

        If TextBoxData = "" Then
            MsgBox("Please Enter your First Name")
        End If
    End Sub

    Private Sub ErrorCheck()

    End Sub
End Class
```

Now, cut the code from your button and paste it between these two new lines, where the cursor is in the image above.

You have just created your own Subroutine!

How to use your new Sub

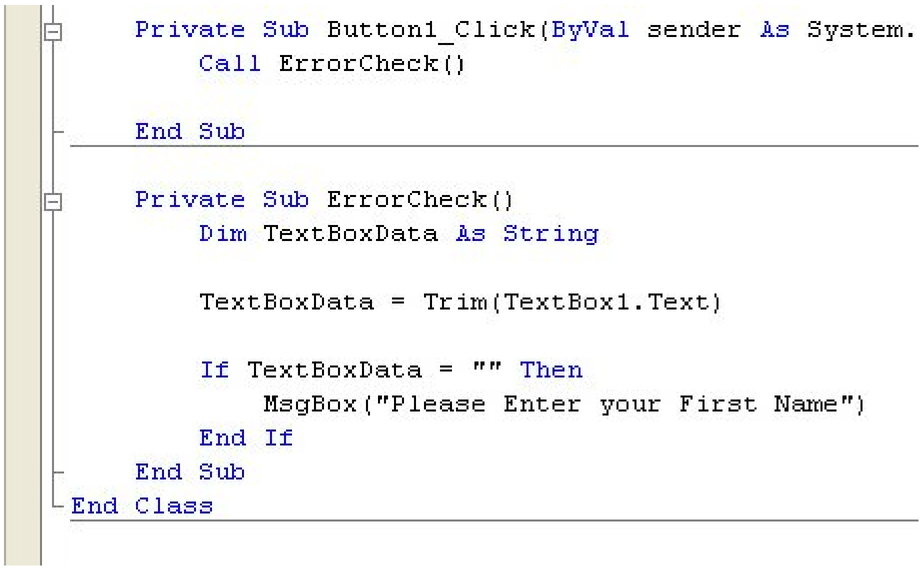
But the Sub is not doing much good where it is. We need a way to use that code. To use our new Sub, we have to tell Visual Basic to execute the code. We do that simply by referring to the Sub by name.

So click inside the button code, just before the End Sub of the button. Type the following:

Call ErrorCheck()

You don't have to use the "Call" word. Try taking that word out altogether, and then testing the programme again. The programme should run exactly the same. But using the "Call" word makes your code easier to read, and tells you that you are executing your own Subroutine on this line.

Your coding window should now look like this:



```
    Private Sub Button1_Click(ByVal sender As System.
        Call ErrorCheck()

    End Sub

    Private Sub ErrorCheck()
        Dim TextBoxData As String

        TextBoxData = Trim(TextBox1.Text)

        If TextBoxData = "" Then
            MsgBox("Please Enter your First Name")
        End If
    End Sub
End Class
```

Run your programme and test it out. You should get the Message Box again, when nothing is in the Textbox.

Add another button to your form. Double click the new button to get at the code. Then type **Call ErrorCheck()** as the code for it. Run your programme again, and click your new button. You should get the Message box popping up, when nothing is entered into the Textbox.

The point about this is that you have created your own code segment. You can use this segment of code whenever you like, just by referring to it by name. Of course, you can have your code check more than one Textbox. You can check as many as you like. And for whatever you like. You can include the code you wrote to check for a correct email address, as well. But all that error checking code is no longer clogging up your button code.

In the next part, we'll take a look at how you can pass text from a text box to your Subroutines - Paramaters, in other words.

Using Parameters in your Subs

This lesson is part of an ongoing tutorial. The first part is here: Create your own Subs in VB .NET

In this lessons, we're going to be exploring Parameters. let's get straight to it.

Add two more textboxes to the form you created in the previous part. then do the following:

- Set the **Name** property of the first Textbox to **txtFirstNumber**
- Set the **Name** property of the second Textbox to **txtSecondNumber**
- Add a new button to your Form and set the **Text** property to "**Get Answer**"

The two textboxes will contain numbers, one for each box. We'll write code to add the two numbers together, but in our own Sub. When the button is clicked, a Message Box will pop up revealing the answer to the sum of the numbers in the textboxes.

Double click your new button to bring up the code window. Click outside of the button code, just after **End Sub**, but before **End Class**. Type the following code:

```
Private Sub AddNumbers()
Dim first As Integer
Dim second As Integer
Dim answer As Integer

first = Val(txtFirstNumber.Text)
second = Val(txtSecondNumber.Text)

answer = first + second

MsgBox("The total is " & answer)
End Sub
```

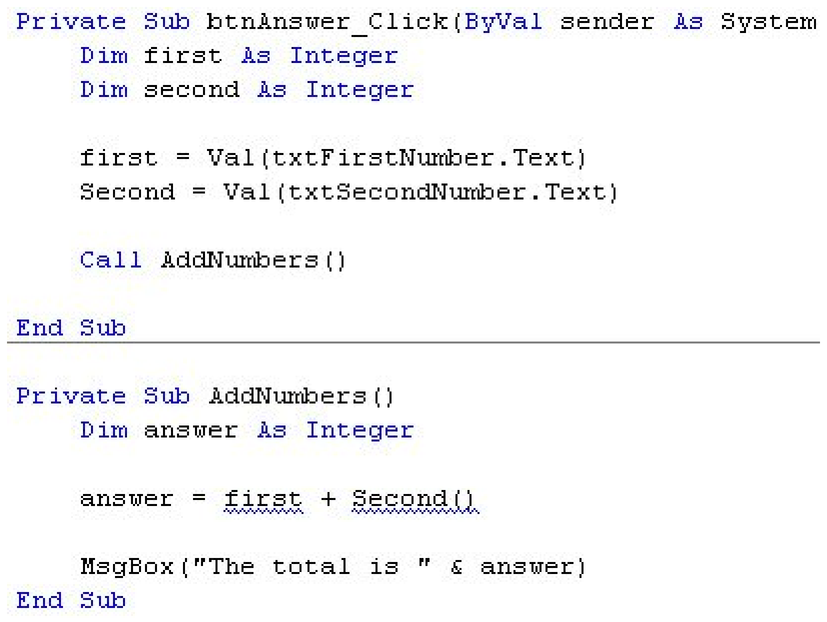
We have created a Sub to add together the two numbers from the Textboxes. The code is very simple, and you should be able to follow it without any problems.

Now add this line to the code for your "Get Answer" button:

```
Call AddNumbers()
```

Run your programme. Type a number in each of the two Textboxes, and click your button to make sure your programme works (did the Message Box display?) Stop the programme and return to the design environment.

Chop the two lines of code for the Textboxes from the Sub and put them into the button. Your two sections of code should now look like this:



```
Private Sub btnAnswer_Click(ByVal sender As System
    Dim first As Integer
    Dim second As Integer

    first = Val(txtFirstNumber.Text)
    Second = Val(txtSecondNumber.Text)

    Call AddNumbers()

End Sub

Private Sub AddNumbers()
    Dim answer As Integer

    answer = first + Second()

    MsgBox("The total is " & answer)
End Sub
```

The reason why there are two wiggly lines under **first** and **Second** is that the AddNumbers Sub knows nothing about these two variables. We've only declared one variable inside the Subroutine - **answer**. To get rid of the wiggly lines, we can set up something called a Parameter. Well, two parameters.

To put it simply, a Parameter is a value that we want to pass from one code section to another. What we want to do is to pass the values we gathered from our button code and hand them over to our **AddNumbers** Sub. So how do we do that?

Change the **Private Sub AddNumbers()** line to this:

Private Sub AddNumbers(first As Integer, second As Integer)

When you press your return key, VB changes the part in round brackets to this:

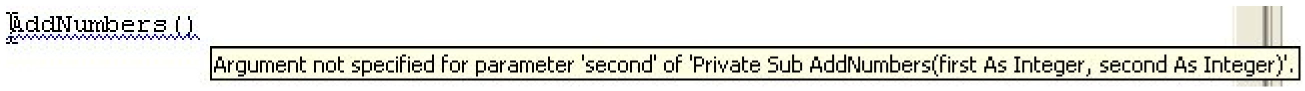
(ByVal first As Integer, ByVal second As Integer)

It's added a curious term - ByVal. We'll explain what that is in a moment. For now, concentrate on the Parameters. The parameters are what we want to hand to our Subroutine. We want to hand an integer variable called **first**, and an integer variable called **second**. Whatever values are currently stored in these two variables will be handed to our Sub.

But we need to change our Calling line, the one from our button. This line now has a wiggly line under it, signifying that something is wrong. Remember, it was this:

Call AddNumbers()

If you hold your mouse over the **AddNumbers**() you might see this tip appear:



```
AddNumbers()
Argument not specified for parameter 'second' of 'Private Sub AddNumbers(first As Integer, second As Integer)'.
```

What this is telling you is that your AddNumbers Sub takes some Parameters (They are called Arguments when you pass them, and Parameters when they are received. Because this is somewhat confusing, we'll stick to calling them Parameters.) In other words, you don't have any option: if you want to call this Sub, you have to add values for the parameters you set up.

So change you Calling line to this:

Call AddNumbers(first, second)

(If the **second** inside your Sub has changed to **Second**(). Delete the round brackets.)

Again, we use the parentheses. And in between the parentheses are our two variables. They don't have to have the same names. Whatever you call your variables in the AddNumbers Sub does not have to be the same names as the calling line. The variable names can be entirely different. But the values in the variables get passed in the order you set them up. In our case the value in the variable **first** will get passed to the first variable in our AddNumbers Sub; the value in the variable **second** will get passed to the next variable we set up in our AddNumbers Sub.

Run your programme and check that it is working properly, now that you have changed the calling line. When you are done, change the variable names for your **AddNumbers** Sub to this:

```
Private Sub AddNumbers(ByVal first2 As Integer, ByVal second2 As Integer)

Dim answer As Integer

answer = first2 + second2

MsgBox "The total is " & answer

End Sub
```

Here, we have changed the names in our Sub. The variable names are now different from the ones in the calling line They are now **first2** and **second2**. But will it still work? Test your programme out and check it. You should find that it does.

So to sum up, we can use a Sub to create our own code segment. We use this Sub just by referring to it by name. If we want to pass any values to our Sub, we can set up Parameters in between the parentheses.

Exercise

Create a Sub to check a Textbox for a valid email address, or adapt the one you already have. Pass whatever is entered in the Textbox to a variable called "email". Pass the value from this variable to your Sub by using a Parameter. When a button is clicked, a message box should pop up telling the user if the email address was wrong.

In the next part, we'll take a look at those two curious terms, **ByVal** and **ByRef**.

ByVal and ByRef in VB .NET

This lesson is part of an ongoing tutorial. The first part is here: Create your own Subs in VB .NET

The word **ByVal** is short for "By Value". What it means is that you are passing a copy of a variable to your Subroutine. You can make changes to the copy and the original will not be altered.

ByRef is the alternative. This is short for By Reference. This means that you are not handing over a copy of the original variable but pointing to the original variable. Let's see a coding example.

Add a new button the form you created in the previous section. Double click the button and add the following code:

```
Dim Number1 As Integer

Number1 = 10
Call IncrementVariable(Number1)

MsgBox(Number1)
```

You'll get a wiggly line under **IncrementVariable(Number1)**. To get rid of it, add the following Subroutine to your code:

```
Private Sub IncrementVariable(ByVal Number1 As Integer)
Number1 = Number1 + 1
End Sub
```

When you're done, run the programme and click your new button. What answer was displayed in the message box?

It should have been 10. But hold on. Didn't we increment the variable Number1 with this line?

Number1 = Number1 + 1

So Number1 started out having a value of 10. After our Sub got called, we added 1 to Number1. So we should have 11 in the message box, right?

The reason Number1 didn't get incremented was because we specified **ByVal** in the Sub:

ByVal Number1 As Integer

This means that only a copy of the original variable got passed over. When we incremented the variable, only the copy got 1 added to it. The original stayed the same - 10.

Change the parameter to the this:

ByRef Number1 As Integer

Run your programme again. Click the button and see what happens.

This time, you should see 11 displayed in the message box. The variable has now been incremented!

It was incremented because we used ByRef. We're referencing the original variable. So when we add 1 to it, the original will change.

The default is **ByVal** - which means a copy of the original variable. If you need to refer to the original variable, use **ByRef**.

In the next part, we'll take a look at Functions in VB .NET

How to Create a Function in VB .NET

This lesson is part of an ongoing tutorial. The first part is here: Create your own Subs in VB .NET

A **function** is more or less the same thing as a Sub - a segment of code you create yourself, and that can be used whenever you want it. The difference is that a Function returns a value, while a Sub doesn't. When you Called a Sub you did this:

```
Call AddNumbers(first, second)
```

Visual Basic will go off and execute that code for you, and then drop down to the next line. The Sub **AddNumbers** is not a value, it's not equal to anything. It's not like a normal variable where you assign something to it. It's just the name of your Subroutine.

A Function is different. It is a value, will be equal to something, and you do have to assign a value to it. You create a Function in the same way you did a Sub, but this time your code will be like this:

```
Private Function ErrorCheck ( ) As Boolean


End Function
```

First, we've changed the word "**Sub**" to "**Function**"; second we've added "**As**" something, in this case "**As Boolean**". The name we called our Function is **ErrorCheck**, and ErrorCheck is now just like a variable. And just like a variable, we use one of the **Types**. We can use "**As Integer**", "**As Long**", "**As Double**", "**As String**", or any of the variable types.

Let's write some code, and try an example.

Add a new button and a textbox to your form. Change the **Name** of the textbox to **txtFunction**. Double click your button and add the following code to it (add it after the **End Sub** of the button, but before the **End Class**):

```
Private Function CheckError ( ) As Boolean

Dim TextBoxData As String

TextBoxData = Trim(txtFunction.Text)

If TextBoxData = "" Then

MsgBox("Blank Text Box detected")

CheckError = True

End If
End Function
```

This is almost the same code from our Sub called ErrorCheck, in a previous section. The difference is the one added line - **CheckError = True**. Remember that **CheckError** is now like a variable. In this case it was a **Boolean** variable. So if there's nothing in the Textbox, we have set **CheckError** to True.

Again, this code is not doing much good by itself. We need a way to use it. This time, because we've set up a Function, we have to assign the value of the function to a variable. Like this:

```
Dim IsError As Boolean

IsError = CheckError ()
```

Here, we are saying "Run the function called **CheckError**. When you have finished, assign the value of **CheckError** to the variable called **IsError**".

Once that code is executed we can then use the variable IsError and test its value. If it's true, then we know that the user did not enter anything into the Textbox; if it's False, then we know that they did. The benefit of using a Function to check for our errors is that we can halt the programme if IsError = True. Like this:

```
If IsError = True then

Exit Sub

End If
```

So double click your button and add the following:

```
Dim IsError As Boolean

IsError = CheckError ()

If IsError = True then

Exit Sub
Else

MsgBox("IsError = False")

End If
```

Run your programme again. Click the button when the textbox is blank, and see what happens. Then enter some text into the textbox, and click your button again.

To sum up, then. A function will return a value. You put this value into the name of your Function. You then assign the value of the Function to a variable. You can then test the variable to see what's in it.

How to use Parameters with Functions

This lesson is part of an ongoing tutorial. The first part is here: Create your own Subs in VB .NET

In the previous section, you learned how to set up a simple function. Let's set up another Function, as a further example. This time we'll add some Parameters to our Function. You use the Parameters in exactly the same way as you did for a Sub.

- So add another button to your Form
- Set its **Text** property to "**Get Function Answer**"
- Add two Textboxes to your Form
- Set the **Name** Property of the first Textbox to **txtNumber1**
- Set the **Name** Property of the second Textbox to **txtNumber2**
- Set up the following Function in your code window (The first line might be spread over two lines here. You can keep yours on one line.)

```
Private Function AddTwoNumbers(ByVal first As Integer, ByVal second As Integer) As Integer

Dim answer As Integer

answer = first + second

AddTwoNumbers = answer

End Function
```

So the name of this Function is **AddTwoNumbers**, and we've set it up to return an **Integer** value. The two parameters we're passing in are also Integers. The code inside the Function simply adds up whatever is inside the variables **first** and **second**. The result is passed to another variable, **answer**. We then pass whatever is inside **answer** to the name of our Function. So **AddTwoNumbers** will be equal to whatever is in the variable **answer**.

Instead of saying AddTwoNumbers = answer you can use the Return keyword. You use it like this:

Return answer

The result is the same: the value inside the variable answer is now the value of the function.

Open up the code for your "Get Answer" button, and add the following code to it:

```
Dim first As Integer
Dim second As Integer
Dim result As Integer

first = Val(txtNumber1.Text)
second = Val(txtNumber2.Text)

result = AddTwoNumbers(first, second)

If result = 0 Then
MsgBox("Please try again ")
Else
MsgBox("The answer is " & result)
End If
```

So we're telling Visual Basic to execute our Function on this line:

result = AddTwoNumbers(first, second)

We're saying, "Run the Function called **AddTwoNumbers**. Hand it the values from the two variables. When you've finished running the function, pass whatever the value of **AddTwoNumbers** is to the variable called **result**."

The next few lines are just testing what is inside the variable result. Remember: the variable **result** will hold whatever the value of **AddTwoNumbers** was.

When you've finished typing your code, run your programme and test it out. Type a number in the first text box, and one in the other. Then click the "Get Function Answer" button. Try typing two zeros into the textboxes and see what happens.

Setting up and using functions can be quite tricky at first, but it's well worth your while persevering: they can vastly improve your coding skills.

In the next section, we explain what a Standard Module is, and how to create your own.

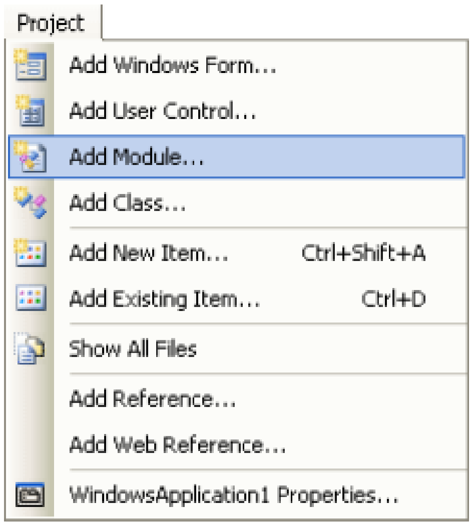
Standard Modules in VB .NET

This lesson is part of an ongoing tutorial. The first part is here: Create your own Subs in VB .NET

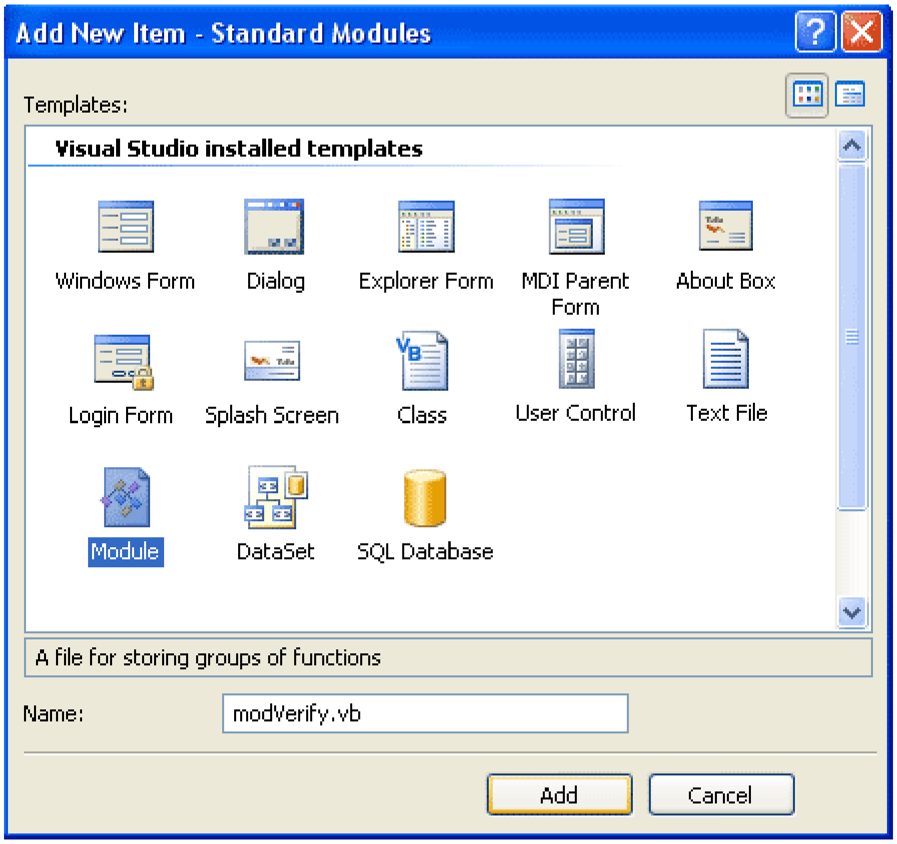
The **Subs** and **Functions** worked perfectly well where they were - inside the two lines "**Public Class Form1**" and "**End Class**". If you tried to put them on a line underneath **End Class** you would get lots of blue wiggly lines under your code.

That's because the code all belongs to Form1. But it doesn't have to. In fact, it's better to separate all your Functions and Subs and put them somewhere else - in something called a **Module**. We'll explore the **Standard Module**, and see how to move our Functions and Subs outside of Form1. That way, we can use them in other projects.

So start a new project. Add a button to you new form. To add a Module to your project, click **Project** from the menu bar. From the from down menu, click on "**Add Module**":



When you click "Add Module", you'll get the following dialogue box popping up:

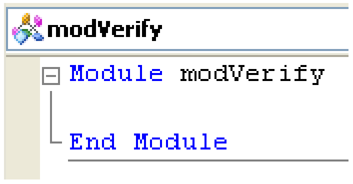


A file for storing groups of functions

Name: modVerify.vb

Select **Module** from the **Templates** window. Type a name for your new module - **modVerify.vb**. When you've typed a name, click the **Open** button.

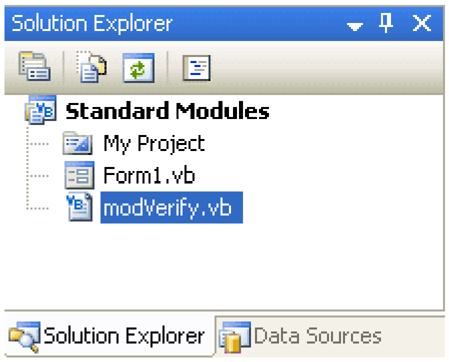
You'll should see a blank window, with this code in it:



modVerify

```
Module modVerify

End Module
```

If you take a look at the **Solutions Explorer** on the right, you should see that your new module is listed:



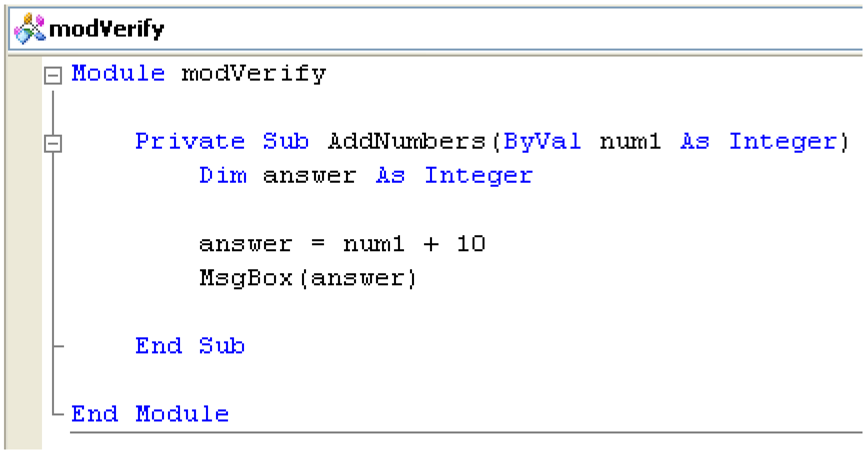
In between **Module modVerify** and **End Module**, type the following Subroutine:

Private Sub AddNumbers(ByVal num1 As Integer)
Dim answer As Integer

answer = num1 + 10
MsgBox(answer)

End Sub

Your coding window should now look like this:



```
Module modVerify

    Private Sub AddNumbers(ByVal num1 As Integer)
        Dim answer As Integer

        answer = num1 + 10
        MsgBox(answer)

    End Sub

End Module
```

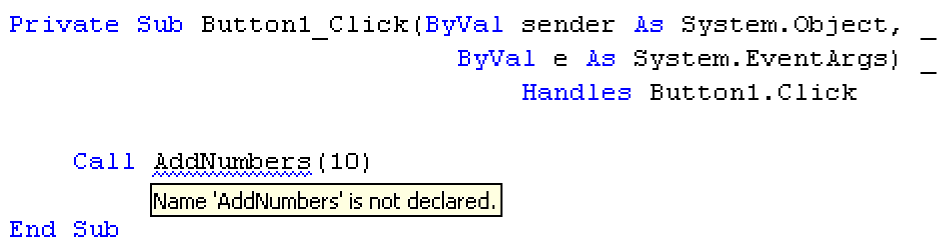
Now click back on your form and double click the button you added. This will bring up the code window for the Form, and the cursor will be flashing inside of the button code.

No more reading these lessons online - get the eBook here!

Add the following code for your button:

Call AddNumbers(10)

When you press the return key, you'll see a blue wiggly line under the Sub name. If you hold your mouse over AddNumbers, you might see this:



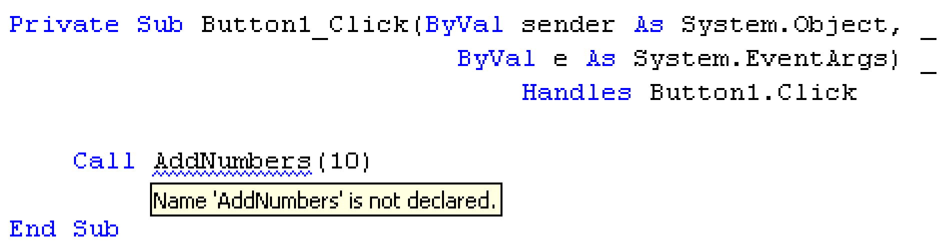
```
Private Sub Button1_Click(ByVal sender As System.Object, _
                            ByVal e As System.EventArgs) _
                                Handles Button1.Click

    Call AddNumbers(10)
         Name 'AddNumbers' is not declared.
End Sub
```

Because this lesson is a bit long, we'll continue it on the next page.

Standard Modules Continued

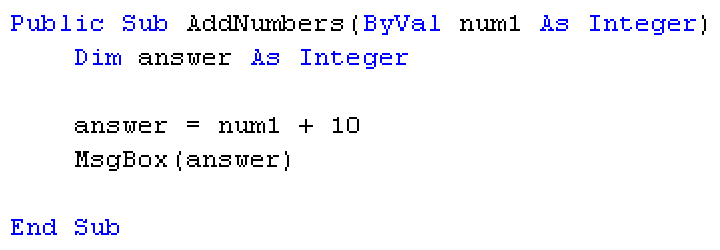
This lesson continues on from the previous part: Standard Modules in VB .NET

In the previous lesson, we saw that we had some problems with our code. VB placed a blue wiggly line under the name of our Sub:



```
Private Sub Button1_Click(ByVal sender As System.Object, _
                            ByVal e As System.EventArgs) _
                                Handles Button1.Click

    Call AddNumbers(10)
         Name 'AddNumbers' is not declared.
End Sub
```

What VB is saying is that it can't see your new Subroutine from the inside of the button code. It thinks you haven't got a Sub called **AddNumbers**. The reason it can't see it is we made the Sub **Private**. Only code inside of the **modVerify** Module can see a Private Sub. If you want the Sub or Function to be available to all the code in your project, including the button, you have to make then Public. This involves nothing more than changing the word **Private** to **Public**. Amend your Sub to this:



```
Public Sub AddNumbers(ByVal num1 As Integer)
    Dim answer As Integer

    answer = num1 + 10
    MsgBox(answer)

End Sub
```

When you make the change from **Private** to **Public**, the blue wiggly line should disappear from the Button code

Run your programme and test it out. Click your Button. You should get a message box saying "20".

We'll now add a Function to our new Module.

So bring up the code for your module. When you have your new Module displayed, type in the following Function:

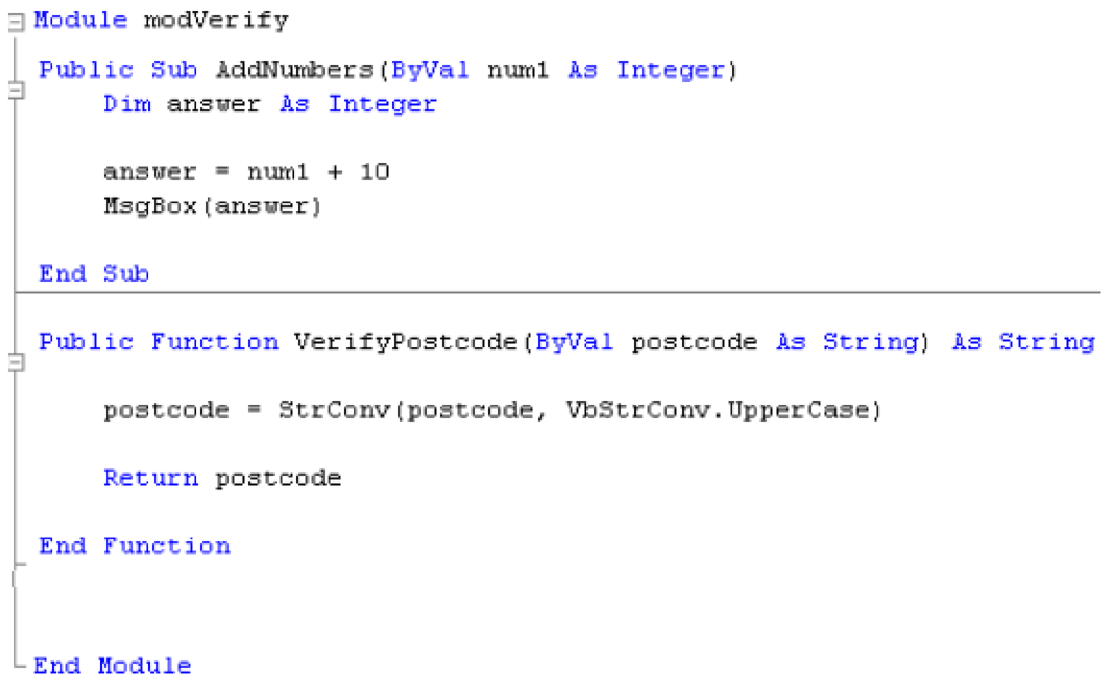
Public Function VerifyPostcode(ByVal postcode As String) As String

postcode = StrConv(postcode, VbStrConv.UpperCase)

Return postcode

End Function

When you're finished, your coding window should look like this:



```
Module modVerify

 Public Sub AddNumbers(ByVal num1 As Integer)
     Dim answer As Integer

     answer = num1 + 10
     MsgBox(answer)

 End Sub

 Public Function VerifyPostcode(ByVal postcode As String) As String

     postcode = StrConv(postcode, VbStrConv.UpperCase)

     Return postcode

 End Function


End Module
```

All the function does is to check a Postcode to see if the letters in it are all in capitals. That's because, quite often, people will enter a postcode as this:

ts1 4jh

What you want is a Postcode that reads:

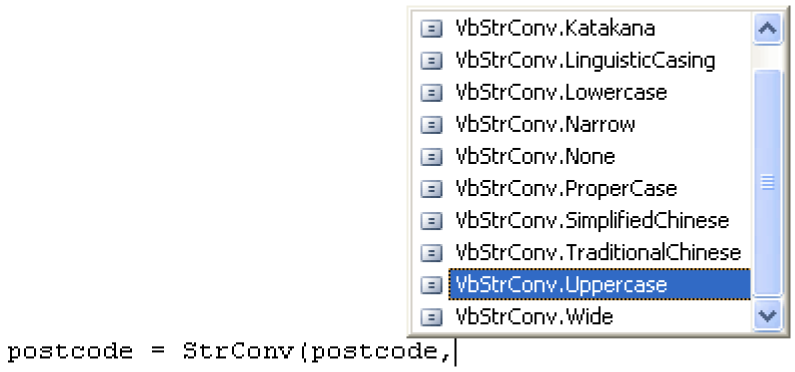
TS1 4JH

The new function will convert a postcode handed to it, and make sure the letters are all capitals.

The inbuilt function that does the converting is worth exploring. It's this:

StrConv( )

This is short for String Conversion. In between the round brackets, VB needs you to put two things: the string you want to convert, and what sort of conversion you want. As soon as you type a comma after the string you want to convert, VB pops up a box of available conversion types:



```
VbStrConv.Katakana
VbStrConv.LinguisticCasing
VbStrConv.Lowercase
VbStrConv.Narrow
VbStrConv.None
VbStrConv.ProperCase
VbStrConv.SimplifiedChinese
VbStrConv.TraditionalChinese
VbStrConv.Uppercase
VbStrConv.Wide
```

```
postcode = StrConv(postcode,
```

A lot on the list are rather baffling. But the one we used was the **UpperCase** one. Simple double click an item to add it to your code. This gave us the following:

StrConv(postcode, VbStrConv.UpperCase)

The function will then convert all the letters in postcode to capitals.

Another useful one on the list is **ProperCase**. What this will do is take a string and convert all the letters of the first word (or words) to capitals. This is useful for addresses. So if somebody entered this as an address:

49 falkland street

The **VbStrConv.ProperCase** item would convert it to this:

49 Falkland Street

But back to our code.

Select your Form again. Then add a new Button, and a Textbox to it. Change the Text property of the Textbox to **ts1 4jh**. Double click the button, and add the following code for it:

```
Dim CheckPostcode As String
Dim ConvertPostcode As String

CheckPostcode = Trim(TextBox1.Text)

ConvertPostcode = VerifyPostcode(CheckPostcode)

TextBox1.Text = ConvertPostcode
```

The first thing we do is get the Text from the textbox. This is passed to a variable called CheckPostcode. The next line calls our new function. We hand it the postcode we got from the textbox:

```
ConvertPostcode = VerifyPostcode(CheckPostcode)
```

When our function has finished the conversion, it will hand back (Return) the result and put it in the variable ConvertPostcode. This is then placed back inside the textbox.

Run your programme, and click the new button. You should find that the letters in the postcode are converted to capitals.

The point about creating a Module to house all your Subs and Functions is that they are in a separate file. You could write more Subs and Functions for your Module, ones that validate text coming from a textbox (an email checker, for example, or one that uses the ProperCase string conversion). You would then have all this code in one file that you could add to totally different projects. If the Subs and Functions were in the same code for the Form, you would have to import the whole Form before you could use the very useful Subs and Functions you created.

But that's enough about Modules. We'll move on to a new section.

The Click Event

What is an Event?

An event is something that happens. Your birthday is an event. So is Christmas. An event in programming terminology is when something special happens. These events are so special that they are built in to the programming language. VB.NET has numerous Events that you can write code for. And we're going to explore some of them in this section.

We'll start with all that mysterious code for the Button's Click Event.

Exploring the The Click Event

Buttons have the ability to be clicked on. When you click a button, the event that is fired is the Click Event. If you were to add a new button to a form, and then double clicked it, you would see the following code stub:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) _
Handles Button1.Click

End Sub
```

This is a Private Subroutine. The name of the Sub is Button1_Click. The Event itself is at the end: Button1.Click. The Handles word means that this Subroutine can Handle the Click Event of Button1. Without the arguments in the round brackets, the code is this:

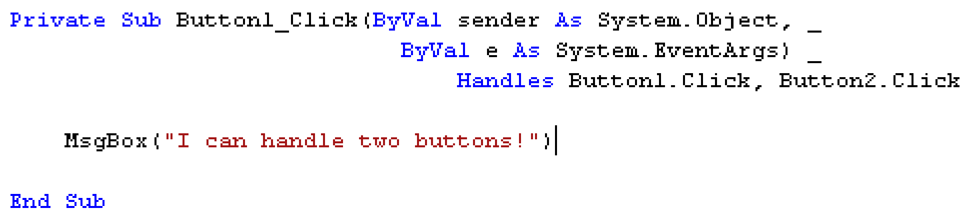
```
Private Sub Button1_Click( ) Handles Button1.Click
```

You can have this Button1_Click Sub Handle other things, too. It can Handle the Click Event of other Buttons, for example. Try this.

- Start a New project
- Give it the name it Events
- When you new Form appears, add two Buttons to it
- Double click Button1 to bring up the code
- At the end of the first line for the Button, add this:

```
Handles Button1.Click, Button2.Click
```

Add a message box as the code for the Button. Your code window might then look like this:



```
Private Sub Button1_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
                             Handles Button1.Click, Button2.Click

    MsgBox("I can handle two buttons!")

End Sub
```

Run your programme, and then click both of the buttons in turn. The same message box appears, regardless of which one you clicked.

The reason it did so was because the Events that the Button1.Click Subroutine can Handle are at the end: the Events for Button1.Click AND Button2.Click.

You can add as many Events as you want on the End. As long as the Subroutine can Handle them, the Event will happen. For example, you could create two more buttons, and then add the Click Event on the end of the first button:

Handles Button1.Click, Button2.Click, Button3.Click, Button4.Click

When you click any of the four button, the code inside of the Button1_Click Subroutine will fire.

However, if you double clicked button2 to try to bring up its coding window, you'd find that the cursor is flashing inside of the code for Button1_Click. Because you've attached the Click Event of button2 to the Button1 Subroutine, you can't have a separate Click Event just for Button2. This Click Event is Handled By the Subroutine called Button1_Click.

Event Arguments

The arguments for a Button's click event, the ones from the round brackets, are these two:

ByVal sender As System.Object, ByVal e As System.EventArgs

This sets up two variable: one called sender and one called e. Instead of sender being an integer or string variable, the type of variable set up for sender is System.Object. This stores a reference to a control (which button was clicked, for example).

For the e variable, this is holding an object, too - information about the event. For a button, this information might be which Mouse Button was clicked or where the mouse pointer was on the screen.

But because this is the Click Event, there's not much more information available: either the button was clicked or it wasn't.

But you can use other Events available to the button. One of these is the **MouseDown** Event. The information for the event would be which button was clicked, where the mouse pointer was when the mouse button was held down, and something called **Delta** (a count of how many notches have been rotated on a mouse wheel).

Let's explore the MouseDown Event.

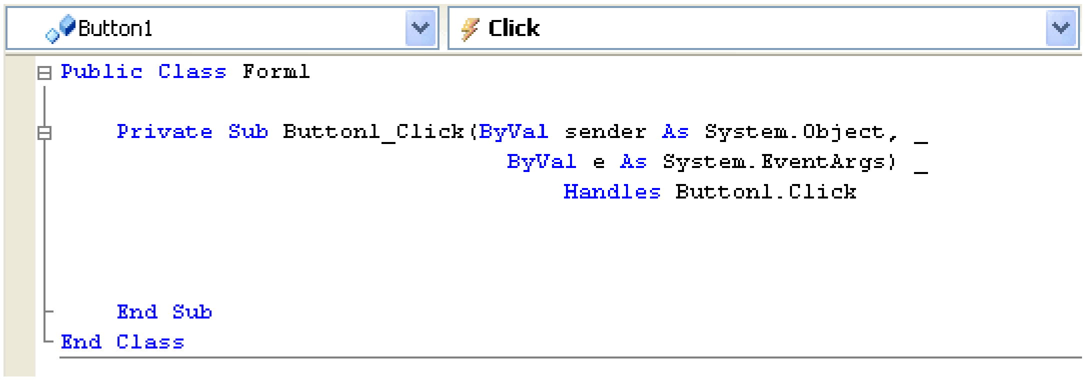
The MouseDown Event in VB .NET

The MouseDown event is available to many controls on the form. A Form can detect when the mouse was held down on it; a textbox can detect when the mouse was held down inside of it; and a Button can detect which mouse button was held down to do the clicking.

We'll see how it all works right now.

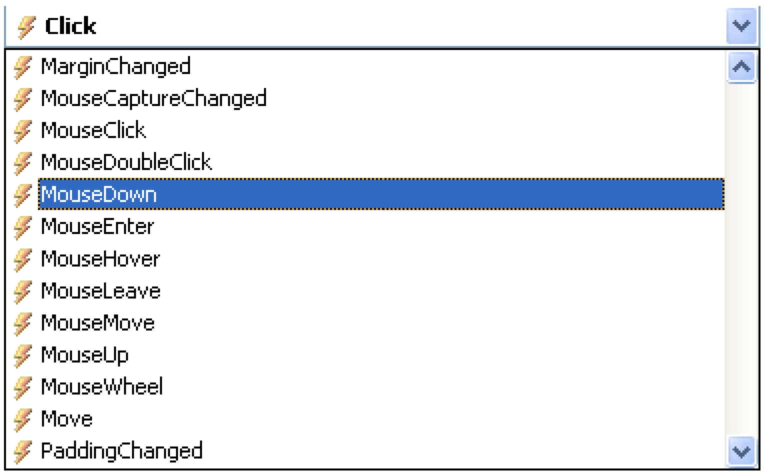
First, delete the all but one of the buttons on your form. (You can right click on a control to delete it. If you haven't been following along from the previous lesson, then just create a new project. Add a Button to your form, and leave iton the default name of Button1.)

Go back to your coding window, and delete any code for the button on your form. Delete any Handles code except for Handles Button1.Click. Your coding window should look something like this one:



```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, _
                              ByVal e As System.EventArgs) _
                                 Handles Button1.Click



    End Sub
End Class
```

Right at the top of the code window, it says **Button1** and **Click**. The lightning bolt next to Click signifies that it is an Event. If you click the drop down box, you'll see a list of other available events:



Scroll down and find the MouseDown event, as in the image above. When you click on it, a new code stub appears, this one (it has been formatted so that the first line is spread over three lines):

```
Private Sub Button1_MouseDown(ByVal sender As Object, _
ByVal e As System.Windows.Forms.MouseEventArgs) _
Handles Button1.MouseDown

End Sub
```

This is a Private Subroutine called **Button1_MouseDown**. Notice that it **Handles** the Button1 **MouseDown** event, and not Button1.Click.

Exploring the Event Arguments

In between the round brackets of the Subroutine, we still have ByVal sender As Object. But we have a new argument now:

ByVal e As System.Windows.Forms.MouseEventArgs

The name of the variable is still **e**. But the type of Object being stored inside of the e variable is different:

System.Windows.Forms.MouseEventArgs

The bit on the end of all that is what we're interested in: MouseEventArgs. This stands for Mouse Events Arguments. What is being stored inside of the e variable is information the Mouse Event: Did you click a button, if so which one?

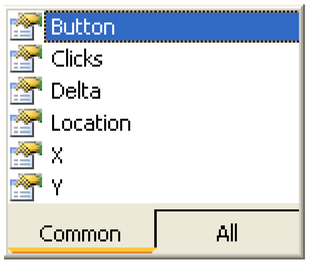
The only thing you need to do to detect which button was pressed is to access a property of the **e** variable. Let's see how to do that.

Which Button was Clicked?

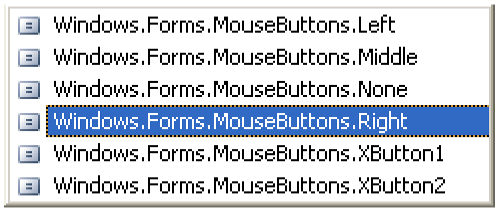
Inside of the Button1_MouseDown Subroutine, type the following code:

If e.Button = MouseButtons.Right Then
MsgBox("Right Button Clicked")
End If

As soon as you type the letter "e", you'll see this pop up box:



To detect which button was clicked, you need the first Property on the list: **Button**. Double click this property to add it to your code. Then after you typed the equals sign, another pop up list appears. This one:


Windows.Forms.MouseButtons.Left
Windows.Forms.MouseButtons.Middle
Windows.Forms.MouseButtons.None
Windows.Forms.MouseButtons.Right
Windows.Forms.MouseButtons.XButton1
Windows.Forms.MouseButtons.XButton2

This is a list of available buttons that VB can detect. Left and Right are the ones you'll use most often.

When you've added the If Statement, your coding window should look something like this:

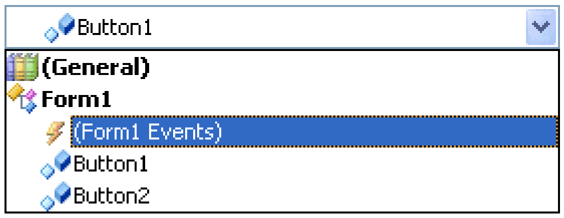
```
Private Sub Button1_MouseDown(ByVal sender As Object, _
                              ByVal e As System.Windows.Forms.MouseEventArgs) _
                                  Handles Button1.MouseDown

    If e.Button = Windows.Forms.MouseButtons.Right Then
        MsgBox("Right button clicked")
    End If

End Sub
```

When you're finished writing your code, run your programme. Click the button with your Left mouse button and nothing will happen. Click it with the Right mouse button and you should see the message box display.

No more reading these lessons online - get the eBook here!

MouseDown and the Form

Stop your programme. When you are returned to the coding environment (Press F7 if you can't see your code), click the down arrow of Button1 at the top of the code. You'll see a drop down box like this:



Select the one highlighted in the image, "Form1 Events". In the Events box to the right, select MouseDown from the list of available events. A new code stub will appear:

```
Private Sub Form1_MouseDown(ByVal sender As Object, _
ByVal e As System.Windows.Forms.MouseEventArgs) _
Handles MyBase.MouseDown

End Sub
```

This time, we have a Private Subroutine called Form1_MouseDown. The two arguments are exactly the same as before. The difference is that now this code Handles the MouseDown event for something called MyBase. (This is an object that refers to the code for Public Class Form1.)

The important thing to bear in mind is that we now have a way to detect when a mouse button was clicked on the form itself.

Add the following code inside of Form1_MouseDown:

```
If e.Button = MouseButtons.Right Then
MsgBox("You clicked on the Form")
End If
```

The only thing that has changed is the Message Box! The If Statement is exactly the same. Run your programme and test it out. Click anywhere on your Form, and you should see the new message box. However, if you right click on the button, you'll get the old message box. Although the button is on the Form, this is considered a separate control from the Form itself. So it has its own events.

You can detect where on the Form the mouse was when the right mouse button was click. Amend your code for Form1_MouseDown. Change it to this:

```
Dim xPos As Integer
Dim yPos As Integer

If e.Button = MouseButtons.Right Then
xPos = e.X
yPos = e.Y
MsgBox("The X Position is " & xPos & " The Y Position is " & yPos)

End If
```

First, we're setting up two integer variable, xPos and yPos. After that we have the same If Statement as before:

```
If e.Button = MouseButtons.Right Then

End If
```

Inside of the If Statement, we're using the X and Y properties of the e variable:

```
xPos = e.X
yPos = e.Y
```

The X property returns how far across, from left to right, the mouse is; the Y property returns how far down, from top to bottom, the mouse is. These values are assigned to our two variables. The result is displayed in a message box.

When you've wrote the code, run your programme and test it out. Right click anywhere on your form. The new message box should display, telling you where the mouse was when the right button was held down.

Click near the top of the form and you'll see the Y position number go down in value; Click near the bottom of the form and you'll see it go up in value. The very top of the form (or a control) has a Y value of zero.

Click from left to right and you'll see the X numbers go up in value. The very left edge of your form has an X value of zero.

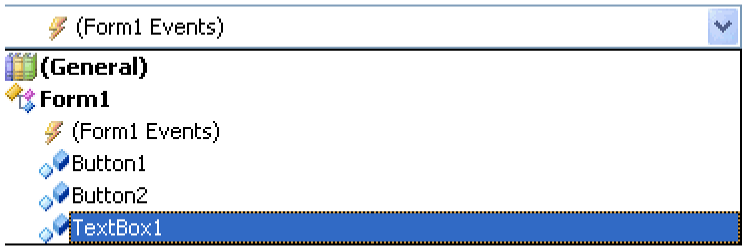
In the next part, we'll explore the KeyDown event.

The KeyDown Event in VB .NET

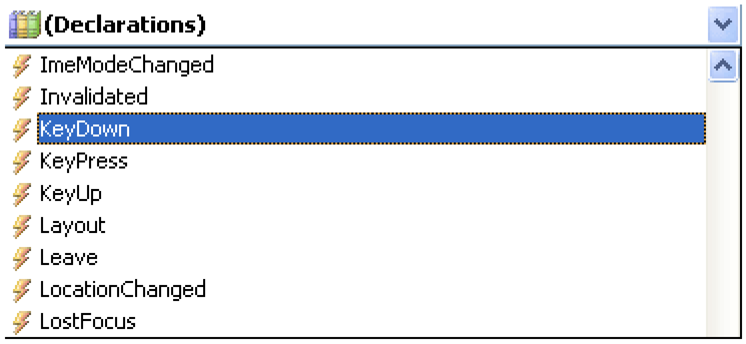
Another useful event is the KeyDown event. As its name suggest, this allows you to detect when a key on the keyboard was held down. This is useful for things like validating text in a textbox.

To test it out, add a textbox to your form. (If you haven't been following the lessons, just start a new project and add a textbox to your new form.). Change the Text property of the textbox to "**Press F1 for help**." Locate the **TabIndex** property in the Property Box, and change it to zero. (The Tab Index sets which control is selected when the Tab key is pressed on the keyboard. By specifying zero as the TabIndex property, you're saying that this should be the first control selected.)

Bring up your code window (you can either click the tab at the top, or press F7 on your keyboard), and click the arrow that reveals the list of controls and objects in your project:



Click on your textbox from the list to select it, as in the image above. Then click the arrow on the Event drop down box to reveal the events available to the textbox. Scroll down and select the KeyDown event:



When you select the KeyDown event, a code stub appears:

```
Private Sub TextBox1_KeyDown(ByVal sender As Object, _
ByVal e As System.Windows.Forms.KeyEventArgs) _
Handles TextBox1.KeyDown
```

End Sub

The event that is being Handled is the **KeyDown** event of TextBox1. Notice, though, that there is a slightly different argument in round brackets:

ByVal e As System.Windows.Forms.KeyEventArgs

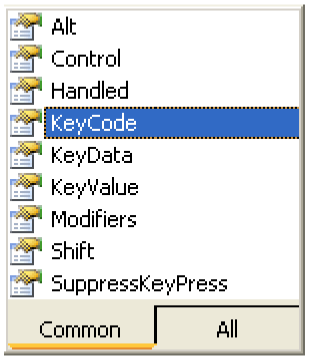
Again, the variable name is still **e**. But now we have something called **KeyEventArgs** on the end. This means that the variable **e** will hold information about the Key on the keyboard that you're trying to detect.

No more reading these lessons online - get the eBook here!

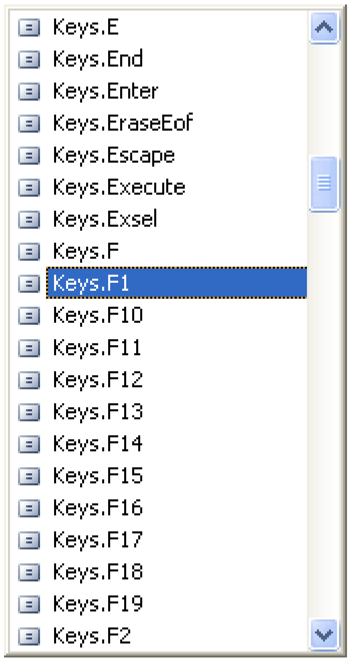
To see what properties the e variable has available to it, add the following to your TextBox1_KeyDown code:

```
If e.KeyCode = Keys.F1 Then
TextBox1.Clear()
MsgBox("Help!!!")
End If
```

As soon as you type the full stop after the letter "e", you'll see this pop up box:



Double click a property to add it to your code. After you type an equals sign, you'll get another pop up box:



The list is a list of keys on your keyboard, some of which you'll have and others that you won't. Scroll down the list until you come to Keys.F1, and double click the item to add it to your code.

The code for the If Statement just clears the textbox and pops up a message.

Try your programme out. Press F1 (If you set TextIndex to zero then the text in the textbox should be selected, and the cursor already flashing. If it's not, click inside of the textbox and then press F1). When the F1 key is pressed, you should see the message box appear.

Another thing you can do is to record the keystrokes a user makes. For example:

```
Dim RecordText as String

RecordText = RecordText & Chr(e.KeyCode)

MsgBox(RecordText)
```

The Chr( ) function converts a KeyCode (which is an integer) to its keyboard character.

But try this exercise.

Exercise

There is an event available to the textbox called **Leave**. Add another textbox to your form, and write code so that the letters in a postcode are converted to uppercase when the user clicks from your first textbox and into your second textbox.

So your first textbox might read "ts1 4jh". When the user clicks inside textbox2, the text from textbox1 should change to "TS1 4JH". The code can be written in the Leave event of textbox1.

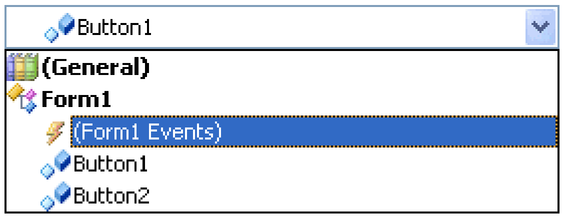
There are an awful lot of Events to explore, and we'll have a look at just one more - the Form Load Event. We'll do that in the next part.

The Form Load Event in VB .NET

An important event you'll want to write code for is the **Form Load** event. You might want to, for example, set the Enabled property of a control to False when a form loads. Or maybe blank out an item on your menu. You can do all this from the Form Load event.

Add another button to your form for this example, and we'll see how the Form Load event works. (If you haven't been folowing along, create a new project and add two buttons to it.

Bring up your coding window, and select the Form1 Events from the drop down box:



In the events drop down box, select Load. A code stub for the Form Load event is then added to your code. Type in the following as the code for the Load Event:

```
MsgBox("Form Load Event")
```

Run your programme. You should see the message box display before the Form loads.

To switch off your second Button before the Form loads, add this to your code:

```
Button2.Enabled = False
```

Run your programme again. You should see that button is no longer available for clicking on.

We'll now up the pace a bit. We'll have a look at just what Objects are, and how you can create your own.

VB.NET Classes and Objects

VB.NET is an Object Oriented programming language. The Objects referred to are created from something called a Class. You've already used Classes throughout this course. But we'll now have a closer look at them.

Object Oriented programming

The modern trend in programming languages is for code to be separated into chunks. When it's being used, each chunk of code (a chunk of code that opens text files, for example) is known as an Object. The code for the Object can then be reused whenever it is needed. Languages like C++ and Java are Object Oriented languages. Until Microsoft came out with VB.NET, the Visual Basic programming language was not OOP (object oriented programming). This time it is.

Object Oriented programming has a steeper learning curve, and things like Encapsulation, Inheritance and Polymorphism have to be digested. We're not going quite that far in this beginner's course. But you should have a good, basic understanding of just what Object are by the end of this section, and how to create your own Objects.

Classes and Objects

In VB.NET, a class is that chunk of code mentioned earlier. You've been using Classes all the time during this course. The Form you've started out with is a Class. If you look right at the top of the code window for a Form, you'll see:

```
Public Class Form1
```

The word "Public" means that other code can see it. Form1 is the name of the Class

If you look at the bottom of the coding window, you'll see End Class, signifying the end of the code for the Class.

When you place a Button or a textbox on the Form, you're really adding it to the Form Class.

When you start the Form, VB does something called instantiation. This basically means that your Form is being turned into an Object, and all the things needed for the creation of the Form are being set up for you (Your controls are being added, variables are being set up an initialised, etc).

And that's the basic difference between a Class and an Object: **A Class is the code itself; the code becomes an Object when you start using it.**

The NET Framework

The NET Framework is something that Microsoft have invested a lot of time, effort and money into creating. It's big. Very big. The way that programming will be done on a Microsoft machine from now on is with NET. And not just on a Microsoft machine. There's something called ADO.NET which is used for creating web site, and for manipulating databases. You can create applications for mobile phones and PDA's with NET. There is even a project in the making that will allow you to write a programme on a Windows machine that will then work on a computer NOT running Windows. All this is made possible with the NET Framework. But what is it?

The NET Framework is a whole lot of Classes (called Namespaces) and the technology to get those Classes to work. The main component is called the Common Language Runtime. A Runtime is the thing that gets your code to actually run on a computer. Previously, these Runtime Languages were machine or programming language specific. The Runtime that gets a Java programme to work, for example, is different to the one that gets a C programme to work. With NET, more than 15 different programming languages can use the Common Language Runtime. One of these languages is, of course Visual Basic NET. Another is C# (pronounce C Sharp). They can all use the Common Language Runtime because of something called the Intermediate Language. (This is a sort of translator for the various languages, and is too advanced to go into for us.)

Namespaces

A Namespace is a group of Classes which are grouped together. The System.IO Namespace you met earlier groups together Classes that you use to read and write to a file. System.Windows.Forms is another Namespace you've met. In fact, you couldn't create your forms without this Namespace. But again, it is just a group of Classes huddling under the same umbrella.

System itself is a Namespace. It's a top-level Namespace. Think of it as the leader of a hierarchy. IO and Windows would be part of this hierarchy, just underneath the leader. Each subsequent group of Classes is subordinate to the one the came before it. For example Forms is a group of Classes available to Windows, just as Windows is a group of Classes available to System. A single form is a Class available to Forms:

System.Windows.Forms.Form

The dot notation is used to separate each group of Classes. A Button is also part of the Forms Class:

System.Windows.Forms.Button

As too is a Textbox:

System.Windows.Forms.TextBox

The leader of the hierarchy is still System, though. Think of it as an army. You'd have a Private who is subordinate to a Sergeant. The Sergeant would be subordinate to a Captain. And the Captain would be subordinate to a General. If the General wanted something done, he might ask the Captain to do it for him. The Captain would get the Sergeant to do it, and the Sergeant would then pick on a poor Private. So Button would be the Private, Forms would be the Sergeant, Windows would be the Captain, and System the General.

In other words, there is a chain of command in NET programming. And if you don't follow the chain of command, you're in trouble!

But you see this chain of command every time you type a full stop and a pop up box appears. When you're selecting an item from the list, you're selecting the next in the chain of command.

This code at the top of the form window:

Inherits System.Windows.Forms.Form

means you don't have to keep typing the full chain of command every time you want to access a button on the form. This chain of command is inherited whenever you create a new VB.NET Form. There are plenty of times when a chain of command is not inherited though, and in that case you do have to type it all out. You did this when you referenced a StreamReader with:

System.IO.StreamReader

The IO Namespace is not inherited when you created a new Form, so you have to tell VB where it is in the hierarchy.

But that's not quite the end of the story. The reason you using all of these long Namespaces is to get at a Property or Method - the chaps that do all the actual work! When you type **Button1.Text**

= **"Click Me"**, Text is a Property of Button1. Button belongs to Form, which belongs to Forms, which belongs to Windows … etc.

So whenever you access a Property or Method of a control, just remember that rather long chain of command.

In the next part, you'll learn how to create your own Classes

Create your own Classes in VB .NET

If you haven't yet read the introduction to Classes, here it is: VB .NET Classes and Objects.

The big benefit of an Object Oriented Programming language is that you can create your own Objects. (It's an Object when you're using the code, remember, and a Class when you're not.)

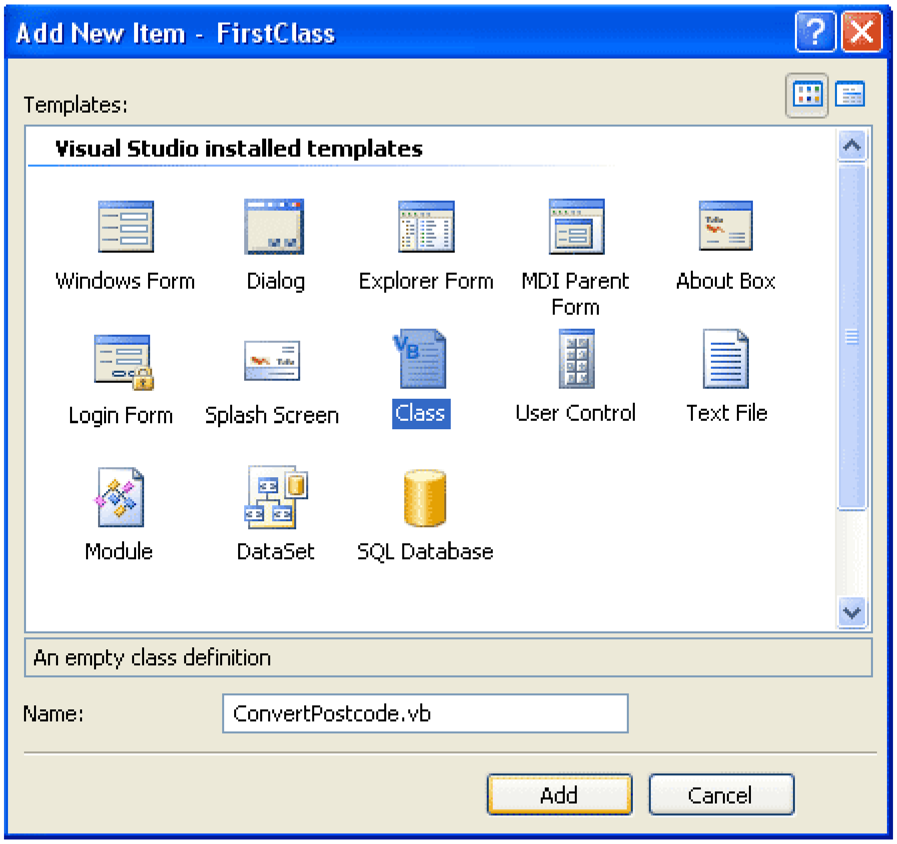
We'll see how to do that now, as we create a very simple Class, and then turn that into an Object.

The Class we'll create is a very simple one, and is intended to show you the basic technique of setting up a Class, then creating an object from it. The Class we'll create will convert the letters in a postcode to uppercase. We'll get the postcode from a textbox on a form. Off we go then.

- Start a new VB .NET project
- Add a Textbox to your form, and leave it on the default **Name**, Textbox1
- Change the **Text** Property to **ts1 4jh** (make sure the letters are lowercase and not upper, because our object will convert it.)
- Add a **Button** to your form

Once you have a new form with a Textbox and a Button on it, you need to add a Class. This is quite easy. It's just like adding a Module. In fact, they look exactly the same!

- So from the VB menu bar, click on **Project**
- From the drop down menu, click **Add Class**
- You'll get this dialogue box popping up:



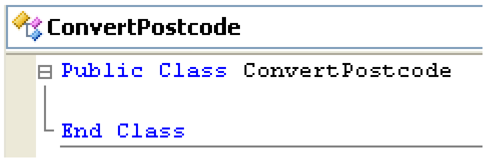
An empty class definition

Name: ConvertPostcode.vb

The Class Template on the right will already be selected. The thing you need to change is the **Name** at the bottom. The default Name is Class1.vb. This is not terribly descriptive, and you'll have great problems working out what this class does, a few months down the line.

Change the **Name** from Class1.vb to **ConvertPostcode.vb**. Then click the **Open** button.

When the code window for the class opens up, it will look like this:



```
Public Class ConvertPostcode

End Class
```

As you can see, there's not a great deal to look at! All we have is the Public Class … End Class code stub. The name of our Class is also there. But the code is in a separate window, and has a tab all to itself. It's this tab full of code that you reuse and turn into an object.

What we have to do now is add the code that does the work - converts our postcode. But we can't just write this:

```
Dim ConvertPostcode As String
ConvertPostcode = StrConv(TextBox1.Text, VbStrConv.UpperCase)
TextBox1.Text = ConvertPostcode
```

That would be all right for the button on our Form. But it's not all right for our Class. When you're designing a Class, your code has to go inside of things like Functions and Subs. You'll also see how to create your own properties for your objects.

When you set up a Function or Sub, you're actually creating Methods for your objects (A Method is code that actually does something, that performs an action. Converts a postcode in our case.) We'll see how to do that next.

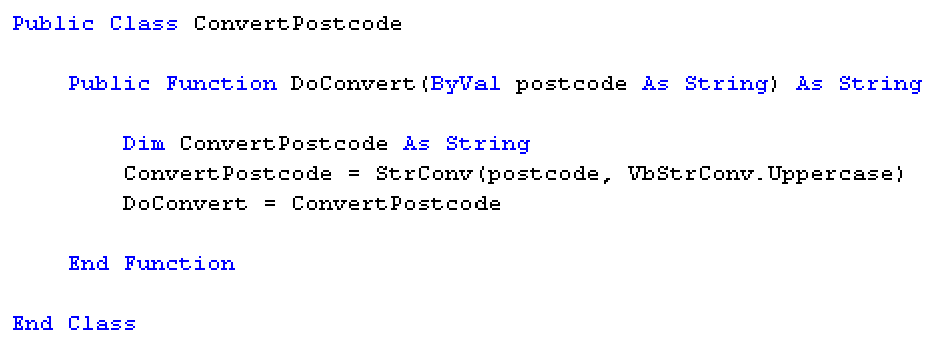
Create Methods in your VB .NET Classes

If you haven't yet read the introduction to Classes, here it is: VB .NET Classes and Objects.

A method created in a Class is nothing more that a Function or a Sub. You've seen how to do this in an earlier section. The process is the same. So add the following code to the Class you created for the previous lesson:

```
Public Function DoConvert(ByVal postcode As String) As String

Dim ConvertPostcode As String
ConvertPostcode = StrConv(postcode, VbStrConv.UpperCase)
DoConvert = ConvertPostcode

End Function
```

When you've finished typing it all, you Class should look like this in the code window:



```
Public Class ConvertPostcode

    Public Function DoConvert(ByVal postcode As String) As String

        Dim ConvertPostcode As String
        ConvertPostcode = StrConv(postcode, VbStrConv.Uppercase)
        DoConvert = ConvertPostcode

    End Function

End Class
```

All we've done is to set up a Public (not private) function. We've given it the name "DoConvert". We've set it up to accept one parameter, a String variable called postcode. This is the value that will be handed to our function. The function itself has been set up as a String. This means that DoConvert will be just like a string variable that we can assign a value to.

The code itself you've met before. It uses the in-built StrConv function to do the actual job of converting the string to uppercase letters.

Now that we've set up a Method, let's see how to create an object from our Class, and put the Method to use.

No more reading these lessons online - get the eBook here!

Creating an Object from a Class

Our function is not much use until we can get it up and running. (Here, "Get it up and running" means create an object from our class.) To do that, double click the button on your Form, and add the following code to it:

```
Dim NewCode As String

Dim objConvertPostcode As ConvertPostcode
objConvertPostcode = New ConvertPostcode

NewCode = objConvertPostcode.DoConvert(TextBox1.Text)

TextBox1.Text = NewCode
```

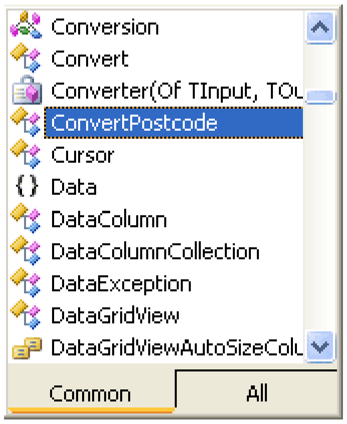
The first line just sets up a new String variable called NewCode. Whatever is returned from our function will be stored inside of this variable.

The next line is where the action starts:

```
Dim objConvertPostcode As ConvertPostcode
```

The variable name **objConvertPostcode** is just something we made up ourselves. The "**obj**" prefix means Object, and this is for our benefit, to remind us that the type of data inside it holds an Object, rather than a plain old Integer or String.

After you type the word "As", then hit your spacebar, you'll see s popup box appear. If you type the letters "conv", you'll see the list automatically move down. The Class your created should be on that list, as in the next image:



You can double click the name of your Class to add it to your code.

But what you're doing in this step is setting up a pointer to your Class. You're telling VB where the Class can be found, and then storing this inside of the variable called **objConvertPostcode**. If VB doesn't know where your Class is then it can't create an Object from it.

The next line of code is the one that creates a new object from your Class:

```
objConvertPostcode = New ConvertPostcode
```

You type the Name of your variable first, then an equals sign. To the right of the equals sign comes the word "New". This is what tells VB to create a New Object. And the class its creating the New Object from is the one called ConvertPostcode.

You can actually type all of that on the same line:

```
Dim objConvertPostcode As ConvertPostcode = New ConvertPostcode
```

This does two jobs: sets a pointer to where the Class is, and creates a new Object from the Class.

But there's reasons why you don't want to do it this way. One is that Objects take up space in memory. And you only really need to create them when they are needed. For example, what if the textbox was blank? You'd want to check for this and invite the user to try again. If you've written everything on one line, then you've already created the Object before the Textbox has been checked. Instead, you could do something like this:

```
If TextBox1.Text = "" Then
MsgBox "Please try again"
Exit Sub
Else
objConvertPostcode = New ConvertPostcode
End If
```

Here's, the textbox is being checked first. If the user has left it blank then we bail out. If not, THEN we create an Object from our Class.

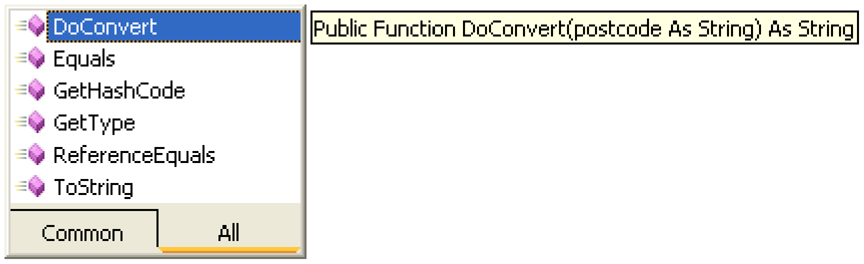
The next line in our code was this:

NewCode = objConvertPostcode.DoConvert(TextBox1.Text)

First, we type the name of the variable we want to store the result of our Method into. In our case, the one called NewCode. After the equals sign, we type the name of our Object variable:

objConvertPostcode

As soon as you type a full stop after your Object variable, you should see a popup box with the name of your new method on the list:



Public Function DoConvert(postcode As String) As String

The image above shows you that the name of our Method "Do Convert" has been recognised by VB. (You can tell it's a Method because of the purple block next to it.) But notice the tool tip - it's the first line from our Function!

In between the round brackets, VB is telling us what type of data needs to be passed over to the Method - a String of text. The second "As String" tells you that the Method returns a value that needs to be stored somewhere.

So if you've set up a Method that returns a value (a Function) then you need to store it in a variable.

To get at the Method inside of your class, first type the name of your Object variable. The type a full stop. Look for the name of your Method in the pop up list that appears.

The final line of the code just assigns the value returned from the Method back to the textbox:

TextBox1.Text = NewCode

Run your code and test it out. Click your Button and you should see the postcode change from "ts1 4jh" to "TS1 4JH".

In the next part, we'll study the subject of creating Methods some more.

Creating Methods that don't return a value

This tutorial follows on from the previous part: Creating Methods. If If you haven't yet read the introduction to Classes, here it is: VB .NET Classes and Objects.

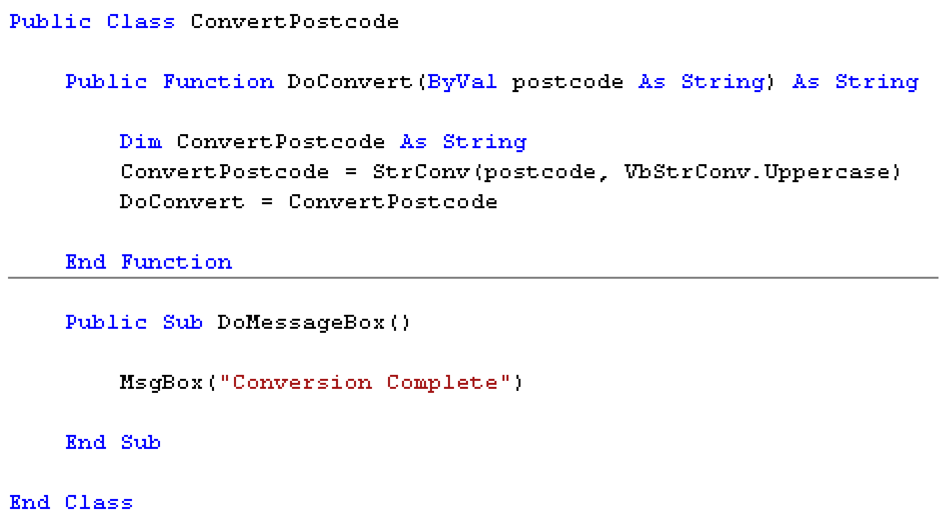
In the last part, you created a Method that returned a value. But you Methods do not have to return a value. In which case, you create a Sub and not a Function, like we did the last time. Add the following code to your Class:

Public Sub DoMessageBox()

MsgBox("Conversion Complete")

End Sub

This is a Public Sub that doesn't do anything terribly useful. But it will illustrate how to write code to set up a Method that doesn't return a value. Here's what your coding window should look like:



```
Public Class ConvertPostcode

    Public Function DoConvert(ByVal postcode As String) As String

        Dim ConvertPostcode As String
        ConvertPostcode = StrConv(postcode, VbStrConv.Uppercase)
        DoConvert = ConvertPostcode

    End Function

    Public Sub DoMessageBox()

        MsgBox("Conversion Complete")

    End Sub

End Class
```

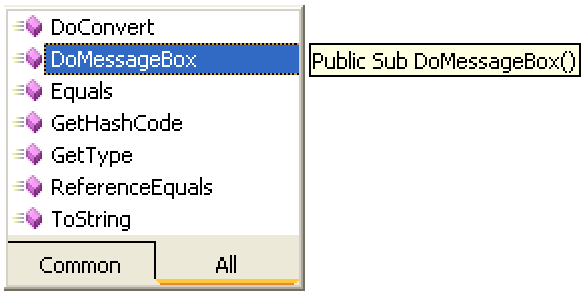
To run your new method, return to your Button code. You should have this, remember:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
                          Handles Button1.Click

    Dim NewCode As String
    Dim objConvertPostcode As ConvertPostcode

    objConvertPostcode = New ConvertPostcode

    NewCode = objConvertPostcode.DoConvert(TextBox1.Text)

    TextBox1.Text = NewCode

End Sub
```

On a new line after **Textbox1.Text = NewCode**, type the following:

objConvertPostcode.DoMessageBox()

After you type the name of your Object variable, then a full stop, you get the message box popping up. This time, your new Method is on the list:



The tip next to the pop up box is telling you that the Method is a Sub, and that it therefore doesn't return a value. The Sub also has no parameters inside of the round brackets, so you don't have to hand anything to it.

You can double click the new Method to add it to your code. When you press the return key on your keyboard, VB adds a pair of round brackets to the end of the line.

So calling a Method that doesn't return a value is just like calling a Sub. The difference is that you need the name of your Object variable first. If we had set up the Sub inside of the Form1 code, we would have just done this to call it:

DoMessageBox()

But because the Sub was inside of the Class, you first type the name of your Object variable:

objConvertPostcode.DoMessageBox()

Run your code again. Click your button. You should see two things happen: one, the text in the textbox will change; and two, you should see message box popping up confirming the change.

Now that you know how to create Methods for your Classes, in the next part you'll see how to create your own Properties.

Creating Properties for your Classes

This tutorial is part of an ongoing lesson. The first part is here: Create your own Classes in VB .NET

In the previous sections, you have learned how to create a Class in VB .NET. You also learned how to create your own Methods. In this part, you'll learn how to create your own properties.

Creating your own Properties in VB .NET Classes

You can add your own Properties to your Class. A Property, remember, is something that changes or sets a value. Examples are, setting the Text in a textbox, changing the background colour of a Form, and setting a Button to be Enabled.

You can Get values from a Property or Set them. So for a Textbox, you can Set the text to appear in the textbox, or you can Get what text is inside of the textbox. You use these same words, Get and Set, when you're creating your own Properties. An example might clear things up.

In this example, we'll allow users to set the height and width of an image. These values will come from textboxes on a form. Before you do the following, download the image you will need for this tutorial:

Download the image for these tutorials (WinZip file)

Once you have the images on your hard drive, do the following:

- Start a new project
- Add a Picture Box control to your Form
- Set the **SizeMode** Property of the Picture box to **StretchImage**
- Click on the **Image** Property, and add the **planet.jpg** image that you downloaded above
- Add two textboxes to the form. Change the Name of the first one to **txtHeight**, and the second one to **txtWidth**. Enter **300** as a the text for both textboxes

- Add two labels to the form. Set the **Text** of the first one to **Height**, and the second one to **Width**. Move them next to the textboxes
- Add a new button to your form. Set the **Text** property to "Change Height and Width"

You now need to add a new class. So from the menu bar, click **Project**. From the **Project** menu, select **Add Class**. Give it the Name of **changeHeightWidth.vb**. Click the **Add** button, and you'll be taken to the coding window for your new Class.

VB now needs to know that you want to set up a Property for your Class. The way you do this is type "**Public Property … End Property**".

On a new line, and before **End Class**, type the following:

```
Public Property ChangeHeight() As Integer
```

ChangeHeight is the name of our property, and it's something we made up ourselves. After a pair of round brackets, you add the type of value that will be returned (Just like a function). Here, we want to return an Integer value.

When you press the return key after typing that line, VB finishes off the rest of the code stub for you:

```
Public Property ChangeHeight() As Integer

Get

End Get

Set(ByVal Value As Integer)

End Set

End Property
```

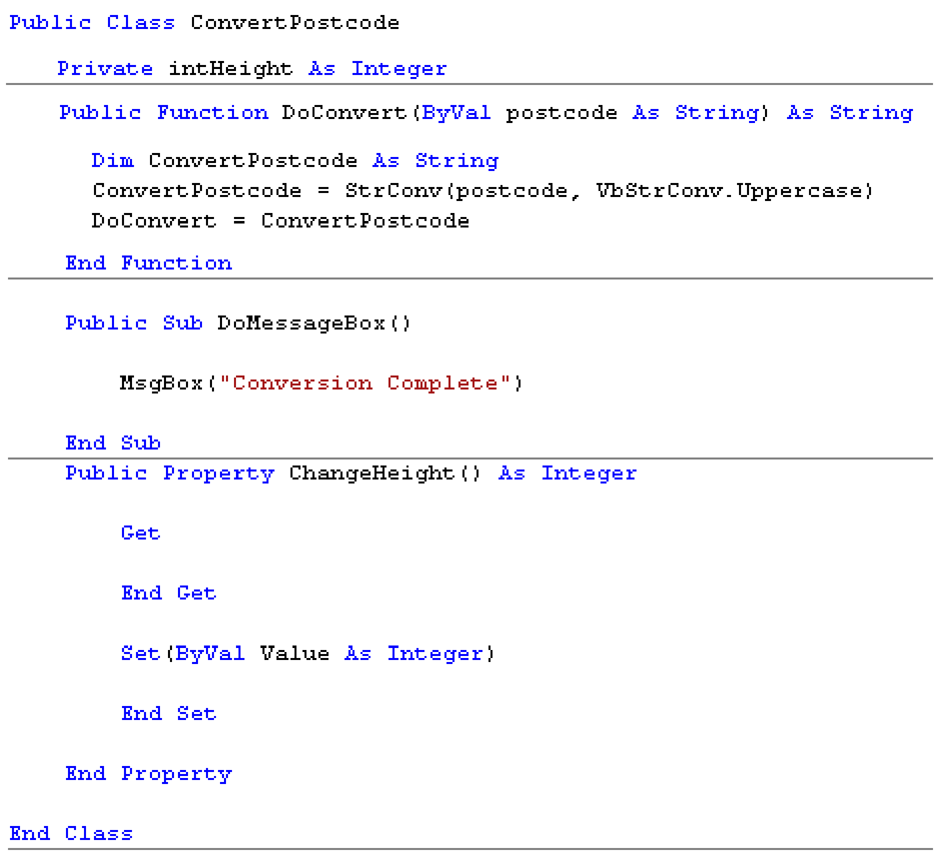
Before the code is explained, add a new variable right at the top of your code, just below "Public Class changeHeightWidth". Add this:

```
Private intHeight As Integer
```

The Private word means that only code inside of the Class can see this variable. You can't access this code directly from a button on a Form, for example.

The reason the variable is right at the top is so that other chunks of code can see and use it.

But your coding window should now look something like this next image:



```
Public Class ConvertPostcode

    Private intHeight As Integer

    Public Function DoConvert(ByVal postcode As String) As String

      Dim ConvertPostcode As String
      ConvertPostcode = StrConv(postcode, VbStrConv.Uppercase)
      DoConvert = ConvertPostcode
    End Function

    Public Sub DoMessageBox()

        MsgBox("Conversion Complete")

    End Sub
    Public Property ChangeHeight() As Integer

        Get

        End Get

        Set(ByVal Value As Integer)

        End Set

    End Property

End Class
```

With the **Get** and **Set** parts, the Property stub is this:

Public Property PropertyName( ) As VaraibleType

End Property

The reason the **Get** and **Set** are there is so that you can Set a value for your property, and get a value back out.

To Set a value, the code inside of Property is this:

Set(ByVal Value As Integer)

End Set

The Set word is followed by a pair of round brackets. Inside of the round brackets is **ByVal Value As Integer**. The is just like a Sub, when you hand over a value to it. The name of the variable, **Value**, is a default name. You can change this to anything you like. The type of variable, **As Integer**, is also a default. You don't have to pass numbers to you property. If you want your Property to handle text you might have something like this:

Set(ByVal MyText As String)

But you couldn't do this:

Set(ByVal Value As Integer, ByVal MyString As String)

In other words, you can't pass two values to your property. You can only pass one value.

But we want to pass a number to our property. For us, this value will come from the textbox on the form. Whatever number is inside of the textbox will get handed over to our Property.

Set(ByVal Value As Integer)

But we need to use this value being handed over. We can assign it to that variable we set up at the top of the Class. So add this to your code (The new line is in bold):

```
Set(ByVal Value As Integer)
intHeight = Value
End Set
```

Whenever our Property is called into action, we're setting a **Value**, and then handing that value to a variable called **intHeight**. This is known as Writing to a Property.

To read from a Property, you use Get. This will Get a value back out of your Property. The code stub is this:

```
Get

End Get
```

You don't need any round brackets for the Get part. You're just fetching something to be read.

Add the line in bold text to your Get statement.

```
Get

ChangeHeight = intHeight

End Get
```

All you're doing here is returning a value, just like you do with a function. You're handing a value to whatever name you called your property. We called ours **ChangeHeight**. It's an Integer. So we can pass whatever value was stored inside of **intHeight** over to the variable called **ChangeHeight**:

ChangeHeight = intHeight

You can also use the Return keyword. Like this:

```
Get

Return intHeight

End Get
```

Let's see how to use our new Property. (It's not a terribly useful property, by the way. A Picture box already has a Height and Width property that you can use. So ours is a bit redundant. But we're keeping it simple so that you can understand how to create your own properties.)

How to use your new Property

This lessons follows on from the previous lesson

The property you have just created is not much good where it is. You need to be able to call it into action. The way you do that is to create a new object from the Class, and then read and write to your property.

So double click the new button you added to your form. Add the following code to it:

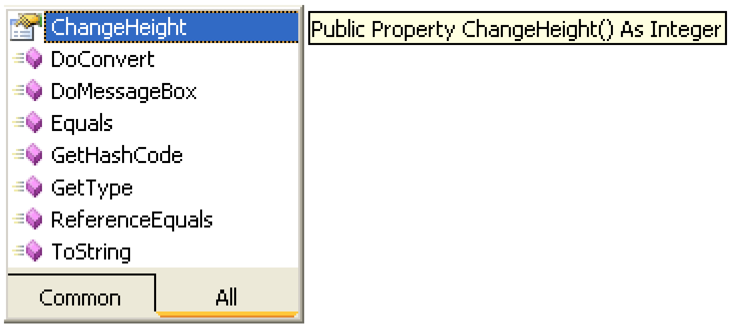
```
Dim objAlterPicBox As changeHeightWidth
Dim NewHeight As Integer

objAlterPicBox = New changeHeightWidth
```

Two of the line you've already met: the one that creates a pointer to a variable (the first line), and the one that creates a new Object from your Class (the third line). The second line just sets up a variable called NewHeight.

To pass a value to your new Property (the Set part), add this line:

```
objAlterPicBox.ChangeHeight = Val(txtHeight.Text)
```

As soon as you type the name of your object variable, then a full stop, you'll see the pop up box appear:



Public Property ChangeHeight() As Integer

The image above is from the VB NET Express edition. Other users may see a different image. But your property will still be on the list. (You can tell it's a Property because the symbol next to it is a hand holding a card.) If you double click the Property, the code is added for you.

After typing an equals sign, you then assign a value to your new property. Here, we're just passing the property whatever value is inside of the textbox called txtHeight.

But we need to Get a value back out, so that we can do something with it. The way you Get at values is to put your code that accesses the property on the right hand side of an equals sign. On the left, You need the name of a variable.

So add the following to your code:

NewHeight = objAlterPicBox.ChangeHeight

So whatever value was stored inside of **ChangeHeight** (or Returned), **Gets** handed over to the **NewHeight** variable.

You can then set the height of the picture box:

PictureBox1.Height = NewHeight

When you add that line above, your Button code should look like this:

```
Private Sub Button2_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
                             Handles Button2.Click

    Dim objAlterPicBox As ConvertPostcode
    Dim NewHeight As Integer

    objAlterPicBox = New ConvertPostcode

    objAlterPicBox.ChangeHeight = Val(txtHeight.Text)
    NewHeight = objAlterPicBox.ChangeHeight

    PictureBox1.Height = NewHeight

    objAlterPicBox = Nothing

End Sub
```

Notice the last line:

objAlterPicBox = Nothing

Because objects take up space in memory, you can release them by setting the object to Nothing. VB is supposed to do the cleaning up for you, but you can't be sure that it's doing its job properly. So it's good form to release your own objects from memory.

When you've finished adding the code, run your programme and test it out. Click your new button and the Height of the picture box should change.

So just to recap:

- You set up a Property by using the following code stub:

Public Property PropertyName( ) As VaraibleType
Get

End Get

Set(ByVal Value As Integer)

End Set
End Property

- The **Set** Statement is for setting values for your properties

- The **Get** Statement is for returning values from your properties
- Once you've created a new object variable to use you class, type the name of the variable and select your property from the pop up list
- When you're setting values for your property, the object variable and property go on the left hand side of the equals sign

ObjectVariableName.PropertyName = PropertyValue

- When you're getting values from your property, the object variable and property go on the right hand side of the equals sign

VariableName = ObjectVariableName.PropertyName

- Release your objects from memory by using the Nothing keyword

OK, that's just about it for this introduction to Classes and Objects. There's an awful lot more to learn about Objects, but as a beginner you have learned the fundamentals.. Before you leave this topic, try this exercise:

Exercise

Set up a property that changes the width of the Picture Box on your Form. Get the new width from the second textbox on your Form.

In the next section, we take a look at VB .NET and Databases.

Visual Basic Express and Databases - the easy way

For this tutorial, we're going to create a simple Address Book project. The names and addresses will come from a Microsoft Access database. Download the database before starting these lessons. Once you have saved the database to your own computer, you can begin.

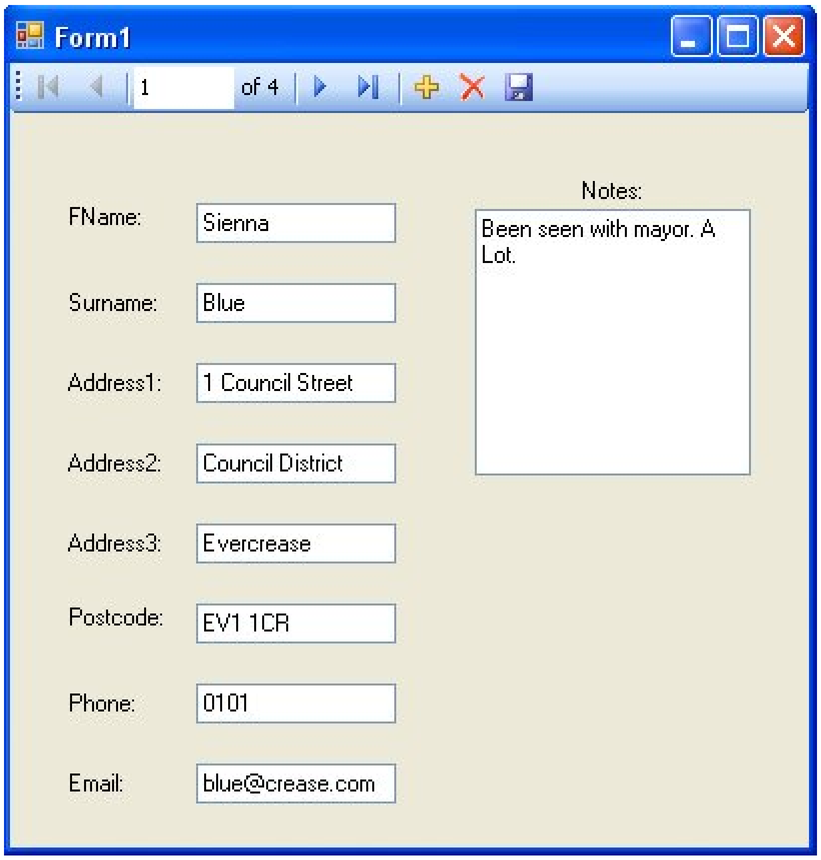
Download the Microsoft Access Database you need for these tutorials

VB.Net allows you many ways to connect to a database or a data source. The technology used to interact with a database or data source is called ADO.NET. The ADO parts stands for Active Data Objects which, admittedly, doesn't explain much. But just like System was a Base Class (leader of a hierarchy, if you like), so is ADO. Forming the foundation of the ADO Base Class are five other major objects:

Connection
Command
DataReader
DataSet
DataAdapter

We'll see just what these objects are, and how to use them, in a later section. But we can make a start on the ADO.NET trail by creating a simple Address Book project. All we'll do is see how to use ADO to open up the database you downloaded, and scroll through each entry.

What we're going to be doing is to use a Wizard to create a programme that reads the database and allows us to scroll through it. The wizard will do most of the work for us, and create the controls that allow users to move through the database. The Form we create will look like this when it's finished:



FName:	Sienna
Surname:	Blue
Address1:	1 Council Street
Address2:	Council District
Address3:	Evercrease
Postcode:	EV1 1CR
Phone:	0101
Email:	blue@crease.com

Notes:
Been seen with mayor. A Lot.

By clicking the buttons at the top, you can scroll through the database in the image above. We'll make a start in the next part.

Click here to start your .NET Database project -- >

<--Back to the .NET Contents Page

View all our Home Study Computer Courses

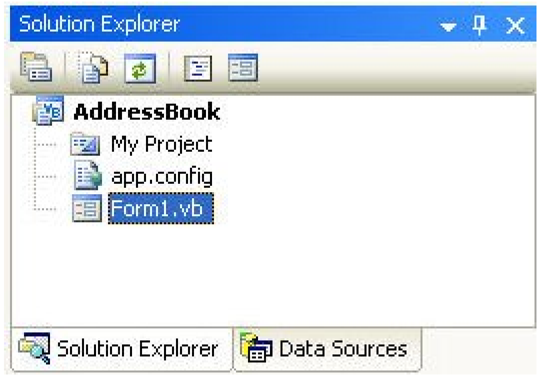
The Database Wizard in VB NET Express

The first part of the tutorial is here: Database project for Visual Basic .NET Express users

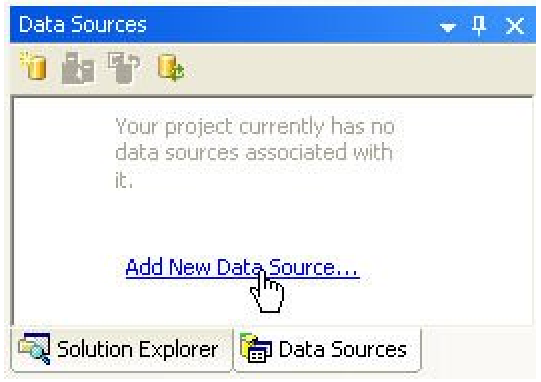
---

Let's make a start on our Database project. So, once you have your VB software open, do the following:

- Click **File** > **New Project** from the menu bar
- Select **Windows Application**, and then give it the **Name** AddressBook. Click OK
- Locate the **Solution Explorer** on the right hand side (If you can't see it, click **View** > **Solution Explorer** from the menu bar, or **View** > **Other Windows** > **Solution Explorer** in version 2010.)

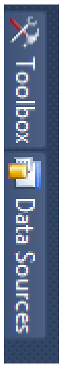


- We need to select a Data Source. So click on **Data Sources** at the bottom of the Solution Explorer in version 2008:

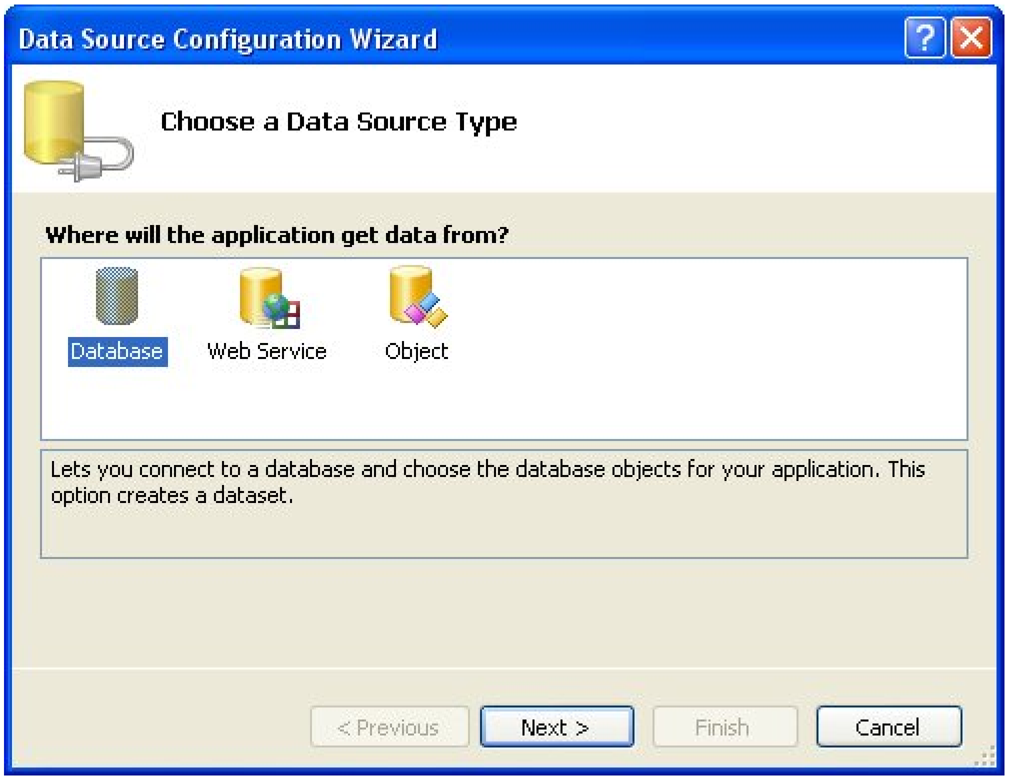


Your project currently has no data sources associated with it.

If you have VB NET 2010 then the Data Source tab is on the left, just below the Toolbox:



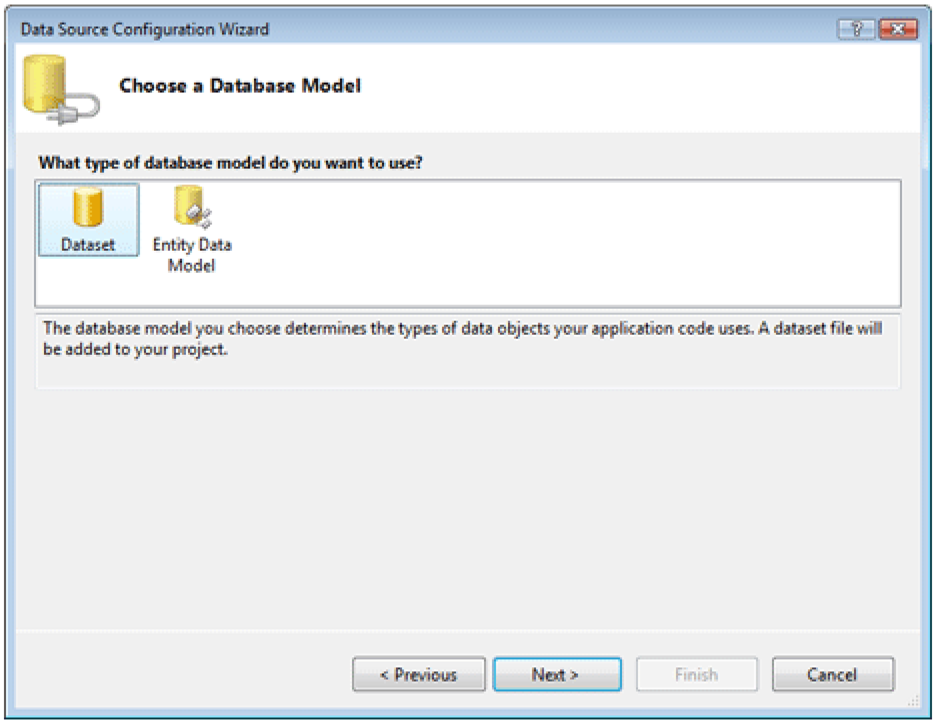
To Add a New Data Source, click on the link. When you do, you'll see a screen welcoming you to the Data Source Configuration Wizard, Just click Next, to get to the screen below:



Where will the application get data from?

Lets you connect to a database and choose the database objects for your application. This option creates a dataset.

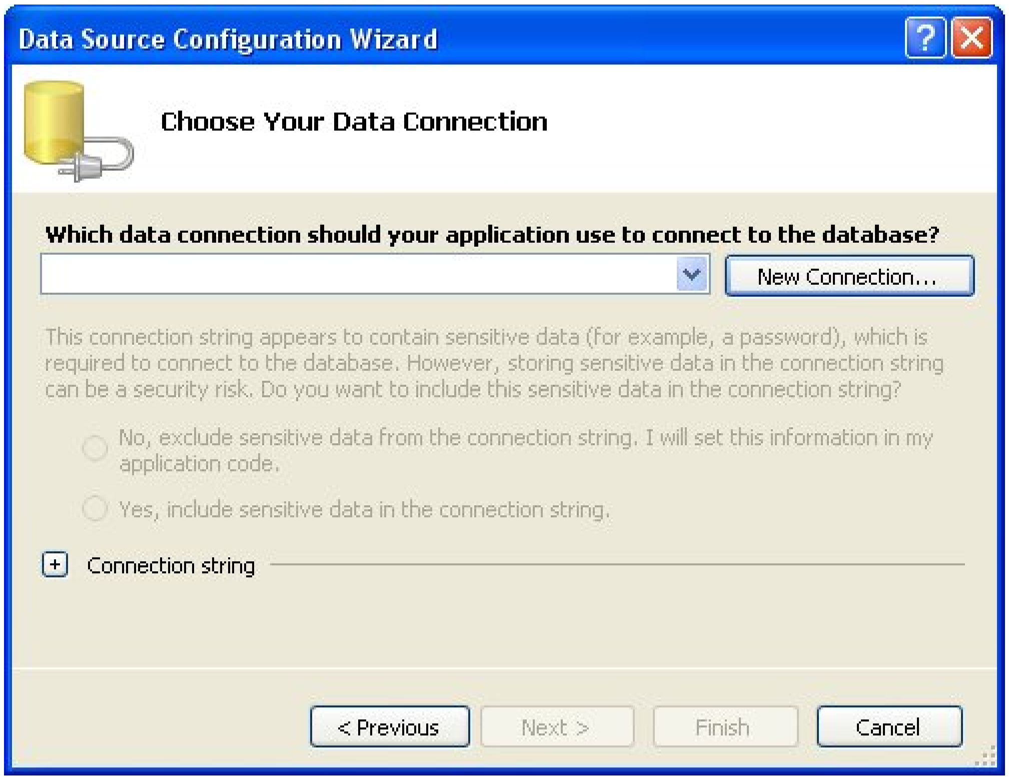
You want to connect to a Database. So select this option, and click Next. In version 2010 of VBN NET, you'll see this screen appear (you won't see it if you have version 2008):


Data Source Configuration Wizard
Choose a Database Model
What type of database model do you want to use?
Dataset
Entity Data Model
< Previous
Next >
Finish
Cancel

The database model you choose determines the types of data objects your application code uses. A dataset file will be added to your project.

Select DataSet and click Next.

In both versions 2008 and 2010, you'll then see this screen:

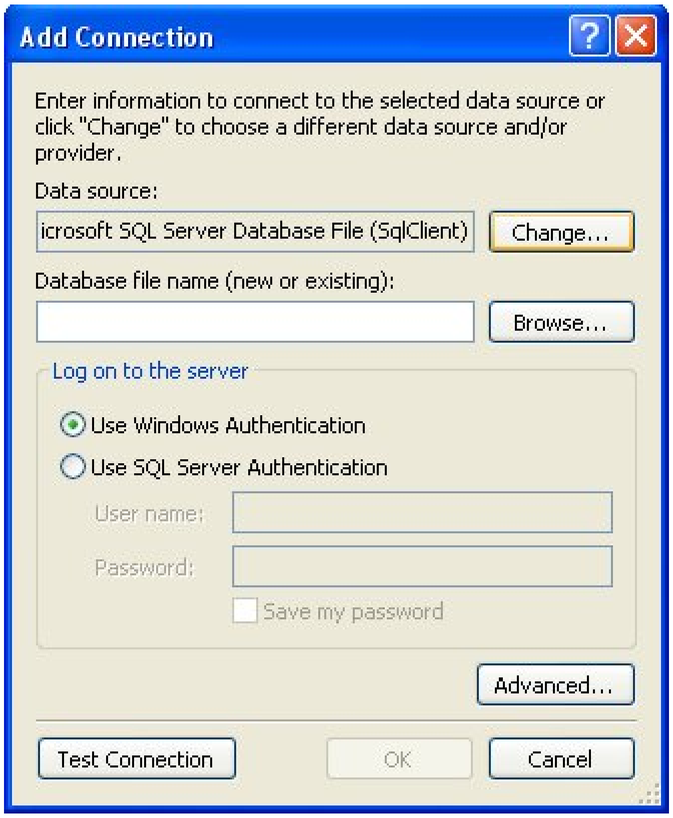

Data Source Configuration Wizard
Choose Your Data Connection
Which data connection should your application use to connect to the database?
New Connection...
Connection string
< Previous
Next >
Finish
Cancel

This connection string appears to contain sensitive data (for example, a password), which is required to connect to the database. However, storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

No, exclude sensitive data from the connection string. I will set this information in my application code.

Yes, include sensitive data in the connection string.

Click the **New Connection** button and another dialogue box pops up:


Add Connection

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source: icrosoft SQL Server Database File (SqlClient)

Database file name (new or existing):

Log on to the server

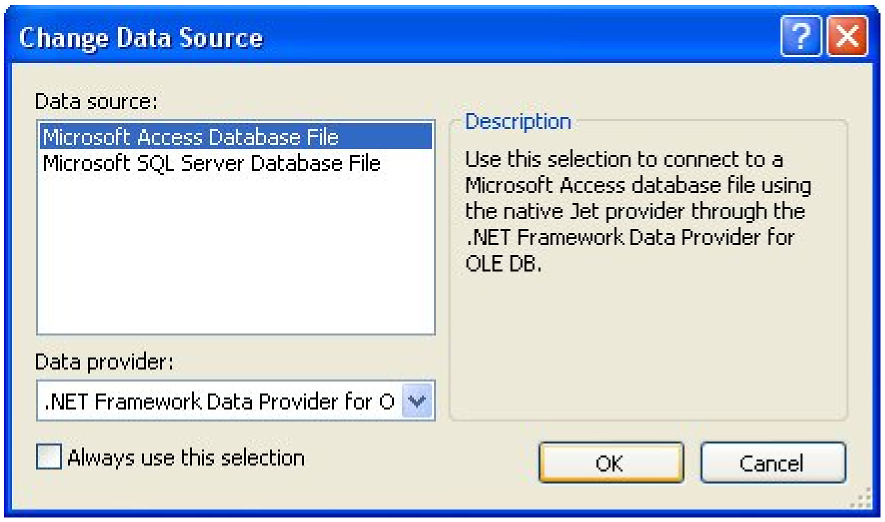
● Use Windows Authentication
○ Use SQL Server Authentication

User name:

Password:

Save my password

Click the **Change** button, because we want to connect to an Access database. (The default is for a SQL Server database.) When you click **Change**, you'll see this:


Change Data Source

Data source:

Microsoft Access Database File
Microsoft SQL Server Database File

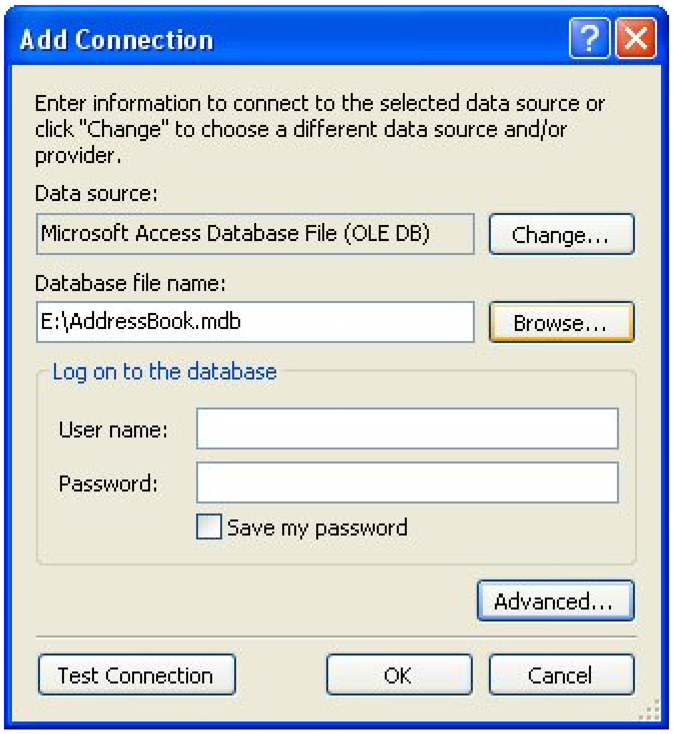
Description

Use this selection to connect to a Microsoft Access database file using the native Jet provider through the .NET Framework Data Provider for OLE DB.

Data provider: .NET Framework Data Provider for O

Always use this selection

Select **Microsoft Access Database File**, then click OK. The previous dialogue box will then look like this:


Add Connection

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source:
Microsoft Access Database File (OLE DB)

Database file name:
E:\AddressBook.mdb

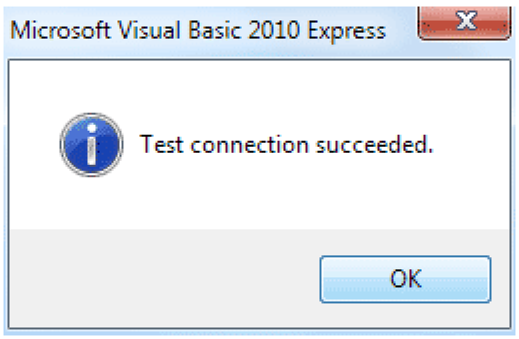
Log on to the database

User name:

Password:

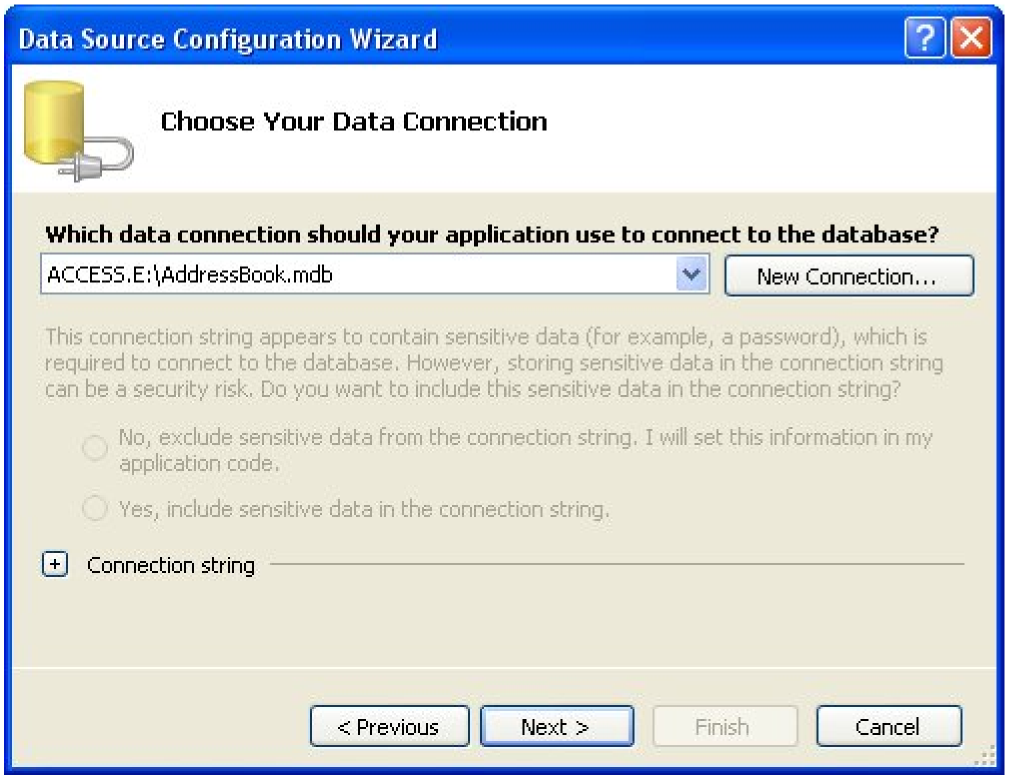
Save my password

Click the **Browse** button and navigate to where on your computer you downloaded our Access Database called AddressBook.mdb. Click **Test Connection** to see if everything is OK, and you'll hopefully see this:


Microsoft Visual Basic 2010 Express

Test connection succeeded.

Click the OK button, then click the OK button on the Add Connection dialogue box as well. You will be returned to the Data Source Configuration Wizard, which should now look like this:



Data Source Configuration Wizard

Choose Your Data Connection

Which data connection should your application use to connect to the database?

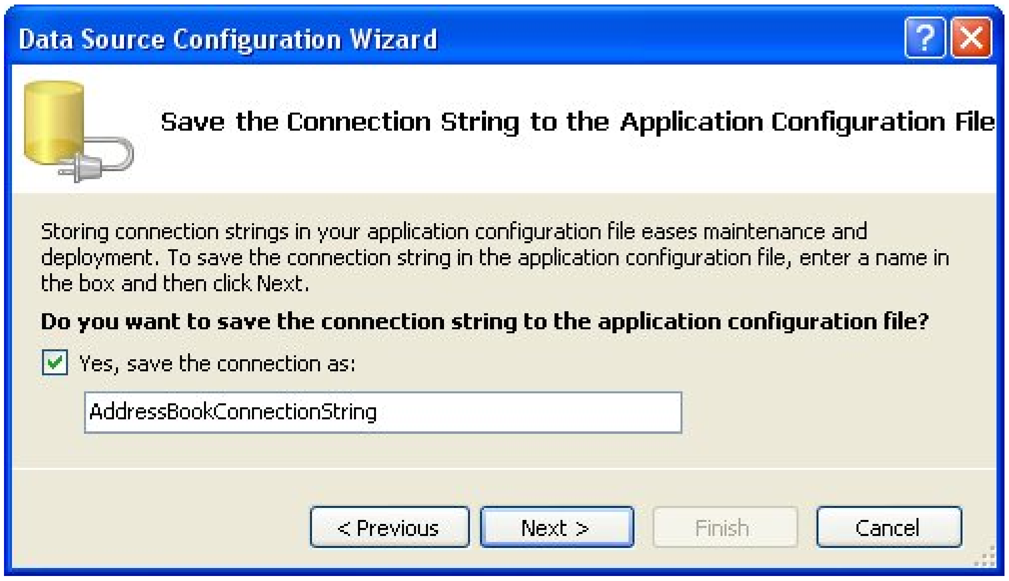
ACCESS.E:\AddressBook.mdb

This connection string appears to contain sensitive data (for example, a password), which is required to connect to the database. However, storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

○ No, exclude sensitive data from the connection string. I will set this information in my application code.

○ Yes, include sensitive data in the connection string.

Click Next to move to the next step of the Wizard. You may see a message box appear, however. Click No on the message box to stop VB copying the database each time it runs. You should then see this:



Data Source Configuration Wizard

Save the Connection String to the Application Configuration File

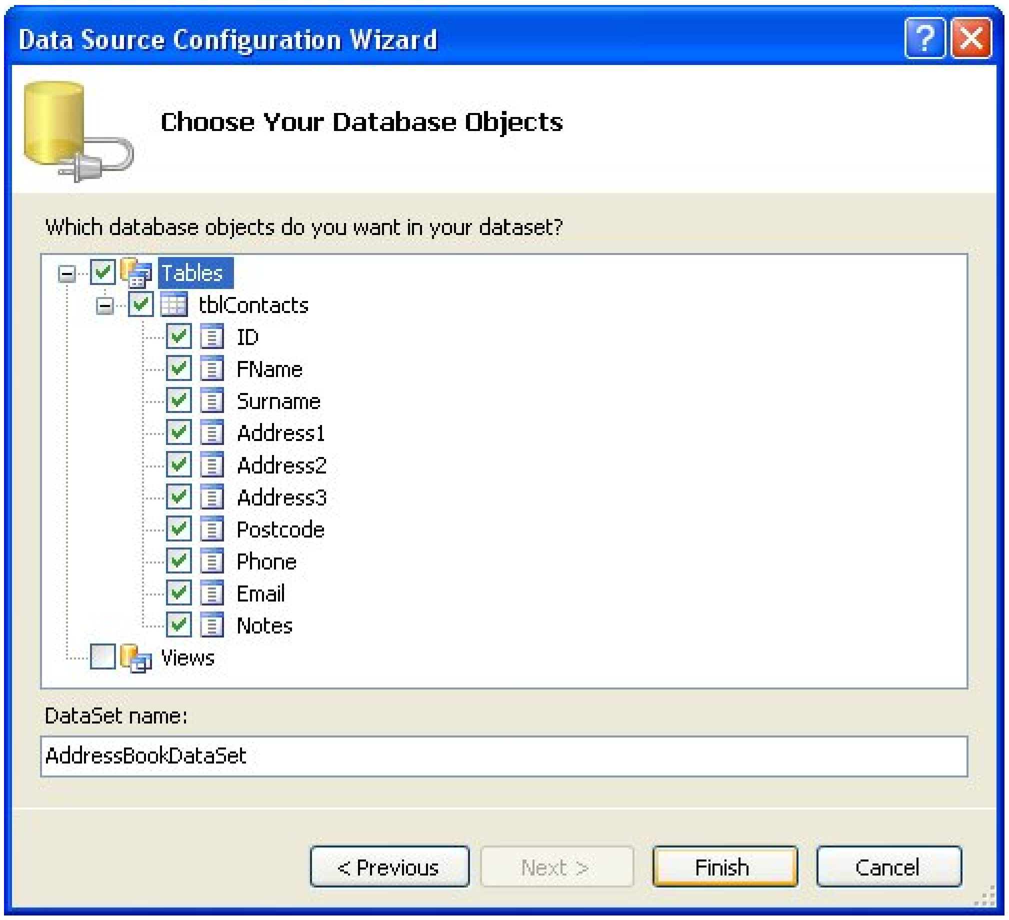
Storing connection strings in your application configuration file eases maintenance and deployment. To save the connection string in the application configuration file, enter a name in the box and then click Next.

Do you want to save the connection string to the application configuration file?

☑ Yes, save the connection as:

AddressBookConnectionString

Make sure there's a tick in the box for "Save the connection", and then click Next:


Data Source Configuration Wizard
Choose Your Database Objects
Which database objects do you want in your dataset?

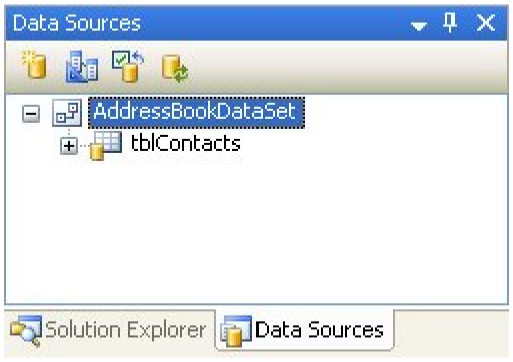
- [✓] Tables
  - [✓] tblContacts
    - [✓] ID
    - [✓] FName
    - [✓] Surname
    - [✓] Address1
    - [✓] Address2
    - [✓] Address3
    - [✓] Postcode
    - [✓] Phone
    - [✓] Email
    - [✓] Notes
- [ ] Views

DataSet name: AddressBookDataSet

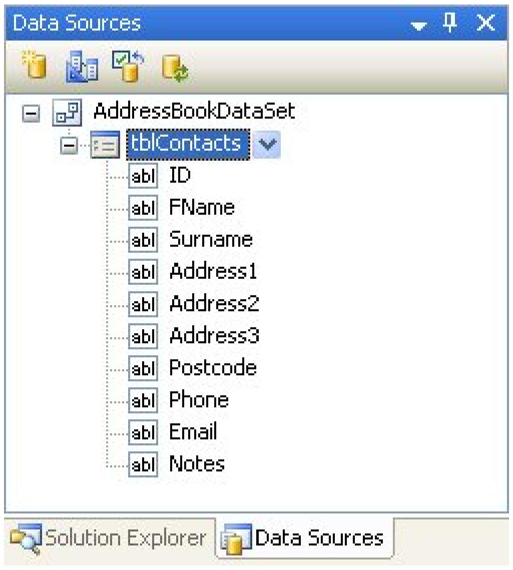
< Previous
Next >
Finish
Cancel

Here, you can select which tables and fields you want. Tick the **Tables** box to include them all. You can give your DataSet a name, if you prefer. Click Finish and you're done.

When you are returned to your form, you should notice the Solution Explorer has added your new Data Source:

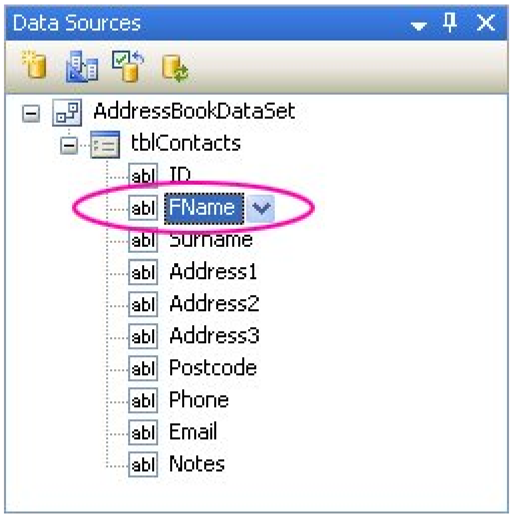


The Data Sources area of the Solution Explorer now displays information about your database. Click the plus symbol (an arrow in VB NET 2010) next to **tblContacts**:



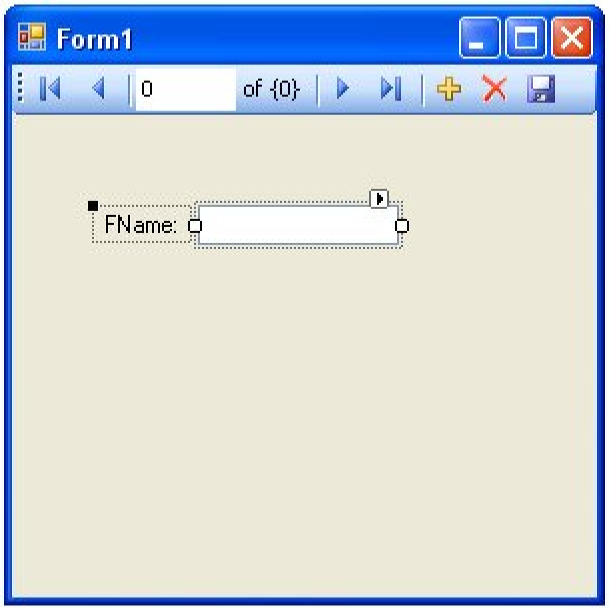
All the Fields in the Address Book database are now showing.

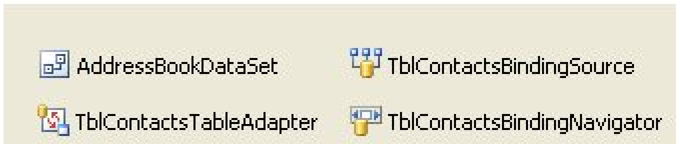
To add a Field to your Form, click on one in the list. Hold down your left mouse button, and drag it over to your form:



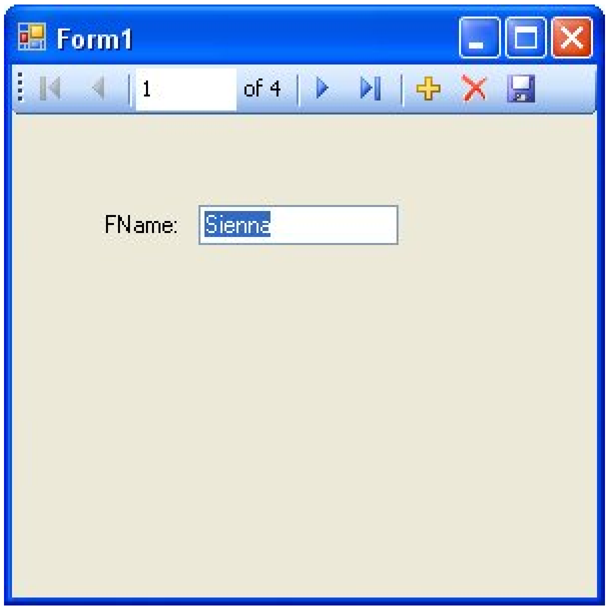
In the image above, the **FName** field is being dragged on the Form. Your mouse cursor will change shape.

When your Field is over the Form, let go of your left mouse button. A textbox and a label will be added. There are two other things to notice: a navigation bar appears at the top of the form, and a lot of strange objects have appeared in the object area at the bottom:



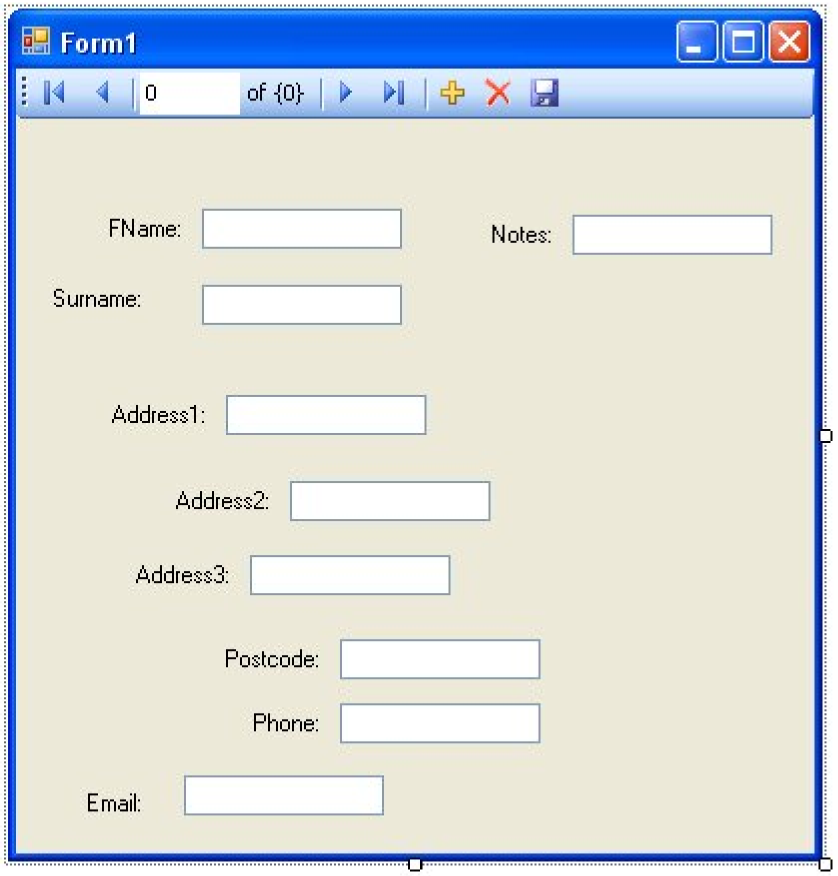


We'll explore the Objects in a later section. But notice the Navigation bar in blue. Run your programme by hitting the F5 key on your keyboard. You should see this:



Click the Navigation arrows to scroll through the database. When you've played around with the controls, stop the form from running, and return to Design View.

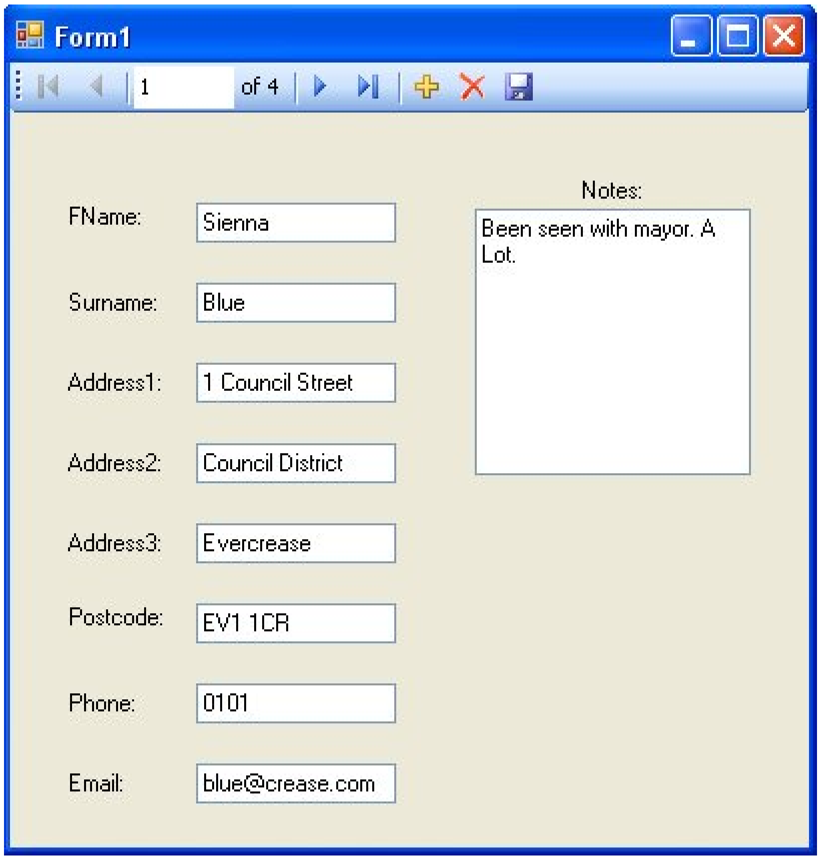
Drag and Drop more Fields to your form. But don't align them yet. We'll see an easy way to do this. But once you've dragged the fields to your form, it might look like this:



I'm sure you'll agree - that's a very untidy form. But there's a very easy way to align all your controls. Try this:

- Click on a Textbox and its label with your left mouse button
- Hold down the Ctrl key on your keyboard, and select a second Textbox and label
- With the Ctrl key still held down, click each Textbox and label in turn
- When all Textbox are selected, click on the **Format** menu at the top
- From the Format menu select **Align** > **Lefts**. The left edges of the Textboxes will align themselves
- From the Format menu select **Vertical Spacing > Make Equa**l. The space between each textbox will then be the same

With your new controls added, and nicely aligned, press F5 to run your form. For the Notes Textbox, set the MultiLine property to True. Your form might then be something like this:



FName: Sienna

Surname: Blue

Address1: 1 Council Street

Address2: Council District

Address3: Evercrease

Postcode: EV1 1CR

Phone: 0101

Email: blue@crease.com

Notes: Been seen with mayor. A Lot.

Click the Navigation icons to move backwards and forwards through your database.

In the next part, you'll move away from the Wizards and learn how to add your own programming code to open up and manipulate databases.

Write your own Database code in VB .NET

In this next section, we'll take a look at the objects that you can use to open and read data from a Database. We'll stick with our Access database, the AddressBook.mdb one, and recreate what the Wizard has done. That way, you'll see for yourself just what is going on behind the scenes.

So close any open projects, and create a new one. Give it whatever name you like, and let's begin.

If you haven't yet downloaded the Address Book database, you can get it here:

Download the Address Book Database

The Connection Object

The Connection Object is what you need if you want to connect to a database. There are a number of different connection objects, and the one you use depends largely on the type of database you're connecting to. Because we're connecting to an Access database, we'll need something called the OLE DB connection object.

OLE stands for Object Linking and Embedding, and its basically a lot of objects (COM objects) bundled together that allow you to connect to data sources in general, and not just databases. You can use it, for example, to connect to text files, SQL Server, email, and a whole lot more.

There are a number of different OLE DB objects (called data providers), but the one we'll use is called "**Jet**". Others are SQL Server and Oracle.

So place a button on your form. Change the **Name** property to **btnLoad**. Double click your button to open up the code window. Add the following line:

```
Dim con As New OleDb.OleDbConnection
```

The variable **con** will now hold the **Connection Objec**t. Notice that there is a full stop after the OleDB part. You'll then get a pop up box from where you can select OleDbConnection. We're also creating a **New** object on this line.This is the object that you use to connect to an Access database.

Setting a Connection String

There are Properties and Methods associated with the Connection Object, of course. We want to start with the ConnectionString property. This can take MANY parameters . Fortunately, we only need a few of these.

We need to pass two things to our new **Connection Object**: the technology we want to use to do the connecting to our database; and where the database is. (If your database was password and user name protected, you would add these two parameters as well. Ours isn't, so we only need the two.)

The technology is called the **Provider**; and you use **Data Source** to specify where your database is. So add this to your code:

Dim dbProvider As String
Dim dbSource As String

dbProvider = "PROVIDER=Microsoft.Jet.OLEDB.4.0;"
dbSource = "Data Source = C:/AddressBook.mdb"

con.ConnectionString = dbProvider & dbSource

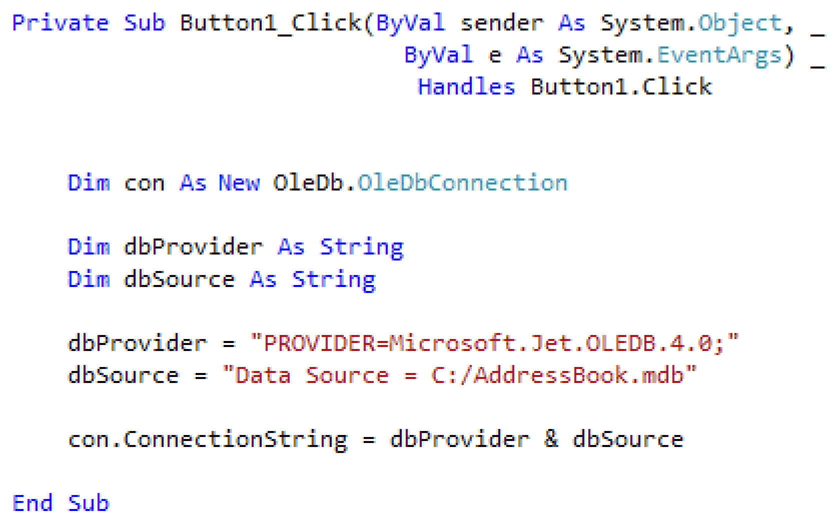
The first part specifies which provider technology we want to use to do the connecting (JET). The second part, typed after a semi-colon, points to where the database is. In the above code, the database is on the C drive, in the root folder. The name of the Access file we want to connect to is called AddressBook.mdb. (Note that "Data Source" is two words, and not one.)

If you prefer, you can have the provider and source on one line, as below (it's on two here because it won't all fit on one line):

con.ConnectionString = "PROVIDER=Microsoft.Jet.OLEDB.4.0;Data Source = C:\AddressBook.mdb"

The first part specifies which provider technology we want to use to do the connecting (**JET**). The second part, typed after a semi-colon, points to where the database is. In the above code, the database is on the C drive, in the root folder. The name of the Access file we want to connect to is called **AddressBook.mdb.** (Note that "**Data Source**" is two words, and not one.)

But your coding window should now look like this:


```
Private Sub Button1_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
                          Handles Button1.Click


    Dim con As New OleDb.OleDbConnection

    Dim dbProvider As String
    Dim dbSource As String

    dbProvider = "PROVIDER=Microsoft.Jet.OLEDB.4.0;"
    dbSource = "Data Source = C:/AddressBook.mdb"

    con.ConnectionString = dbProvider & dbSource

End Sub
```

This assumes that you have copied the AddressBook database over to the root folder of your C Drive. If you've copied it to another folder, change the "Data Source" part to match. For example, if you copied it to a folder called "databases" you'd put this:

Data Source = C:\databases\AddressBook.mdb

You can also specify a folder such as MyDocuments (or Documents in Vista and Windows 7). You do it like this:

dbSource = "Data Source = C:\Users\Owner\Documents\AddressBook.mdb"

Another way to specify a file path is this:

```
Dim fldr As String
fldr = Environment.GetFolderPath( Environment.SpecialFolder.MyDocuments ) &
"/AddressBook.mdb"

dbSource = "Data Source = " & fldr
```

On the second line, spread over two lines in the code above, we have this:

Environment.GetFolderPath( )

The folder path you're getting goes between the round brackets of GetFolderPath:

Environment.SpecialFolder.MyDocuments

The Special Folder in this case is the MyDocuments folder.

But back to our connection code. **ConnectionString** is a property of the **con** variable. The con variable holds our Connection Object. We're passing the Connection String the name of a data provider, and a path to the database.

Opening the Connection

Now that we have a ConnectionString, we can go ahead and open the datatbase. This is quite easy - just use the **Open** method of the Connection Object:

con.Open( )

Once open, the connection has to be closed again. This time, just use the Close method:

con.Close( )

Add the following four lines to your code:

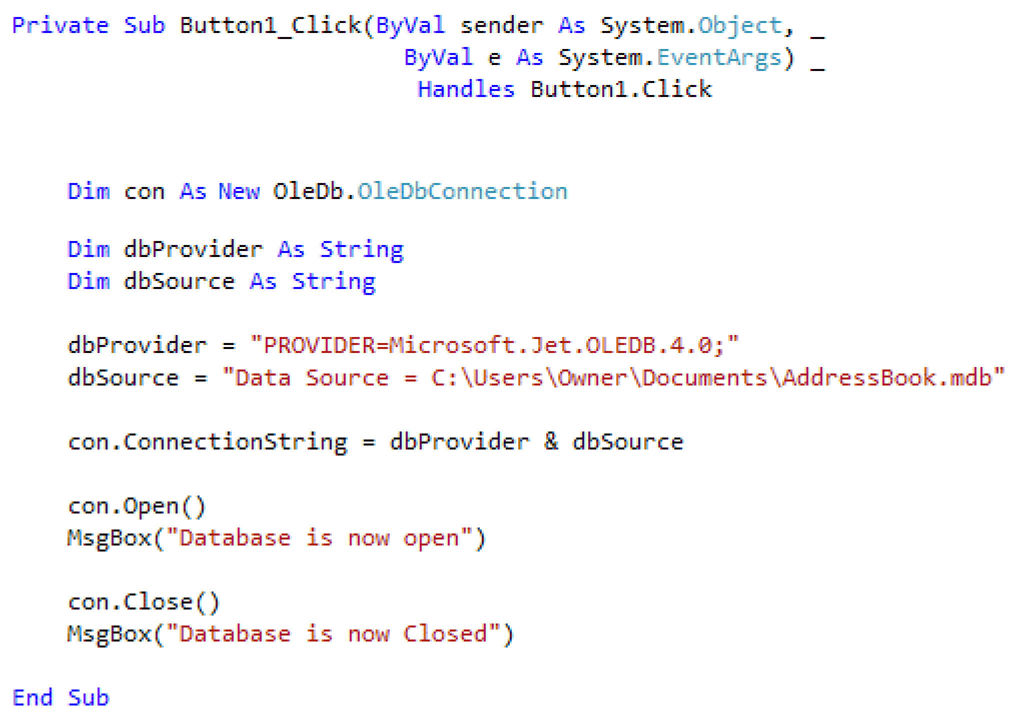
con.Open()

MsgBox("Database is now open")

con.Close()

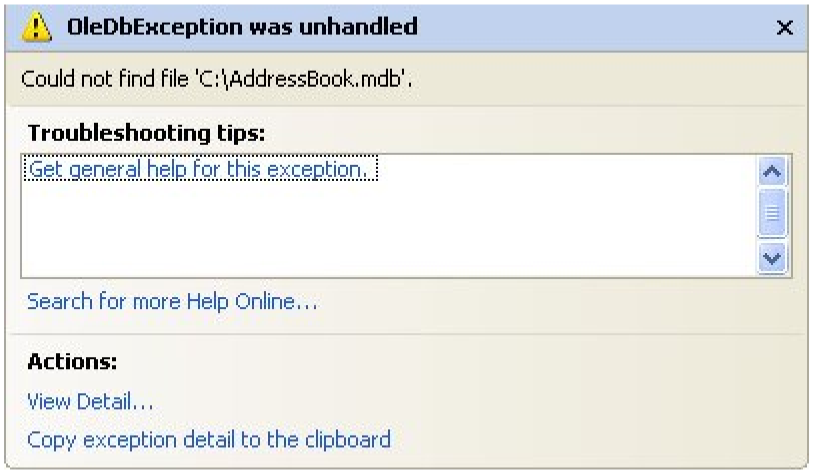
MsgBox("Database is now Closed")

Your coding window will then look like this (use the file path below, if you have Vista or Windows 7, after moving the database to your Documents folder):



```
Private Sub Button1_Click(ByVal sender As System.Object, _
                         ByVal e As System.EventArgs) _
                          Handles Button1.Click


    Dim con As New OleDb.OleDbConnection

    Dim dbProvider As String
    Dim dbSource As String

    dbProvider = "PROVIDER=Microsoft.Jet.OLEDB.4.0;"
    dbSource = "Data Source = C:\Users\Owner\Documents\AddressBook.mdb"

    con.ConnectionString = dbProvider & dbSource

    con.Open()
    MsgBox("Database is now open")

    con.Close()
    MsgBox("Database is now Closed")

End Sub
```

Test out your new code by running your programme. Click your button and the two message boxes should display. If they don't, make sure your Data Source path is correct. If it isn't, you might see this error message:


OleDbException was unhandled

Could not find file 'C:\AddressBook.mdb'.

Troubleshooting tips:

Get general help for this exception.

Search for more Help Online...

Actions:

View Detail...

Copy exception detail to the clipboard

The error message is a bit on the vague and mysterious side. But what it's saying is that it can't find the path to the database, so it can't Open the connection. The line con.Open in your code will then be highlighted in green. You need to specify the correct path to your database. When you do, you'll see the message boxes from our code, and not the big one above.

Now that we've opened a connection to the database, we need to read the information from it. This is where the DataSet and the DataAdapter come in.

Data Sets and Data Adapters

The first part of Databases and VB .NET can be found here:

Coding your own VB .NET database projects

In the previous part, you learned how to set up a Connection Object. This was so that you could open a connection to the database itself. But that's not the end of it. The data from the database needs to be stored somewhere, so that we can manipulate it.

ADO.NET uses something called a **DataSet** to hold all of your information from the database (you can also use a DataTable, if all you want to do is read information, and not have people write to your database.). But the **DataSet** (and Data Table) will hold a copy of the information from the database.

The DataSet is not something you can draw on your form, like a Button or a Textbox. The DataSet is something that is hidden from you, and just stored in memory. Imagine a grid with rows and columns. Each imaginary row of the DataSet represents a Row of information in your

Access database. And each imaginary column represents a Column of information in your Access database (called a Field in Access).

This, then, is a DataSet. But what's a Data Adapter?

The Connection Object and the DataSet can't see each other. They need a go-between so that they can communicate. This go-between is called a Data Adapter. The Data Adapter contacts your Connection Object, and then executes a query that you set up. The results of that query are then stored in the DataSet.

The Data Adapter and DataSet are objects. You set them up like this:

```
Dim ds As New DataSet
Dim da As OleDb.OleDbDataAdapter

da = New OleDb.OleDbDataAdapter(sql, con)
```

The code needs a little explaining, though. First, the Data Adapter.

The Data Adapter

The Data Adapter is a property of the OLEDB object, hence the full stop between the two:

OleDb.OleDbDataAdapter

We're passing this object to the variable called **da**. This variable will then hold a reference to the Data Adapter.

While the second line in the code above sets up a reference to the Data Adapter, the third line creates a new Data Adapter object. You need to put two things in the round brackets of the Object declaration: Your SQL string (which we'll get to shortly), and your connection object. Our Connection Object is stored in the variable which we've called **con**. (Like all variable you can call it practically anything you like. We've gone for something short and memorable.) You then pass the New Data Adapter to your variable (**da** for us):

da = New OleDb.OleDbDataAdapter(sql, con)

We need something else, though. The **sql** in between the round brackets is the name of a variable. We haven't yet set this up. We'll have a look at SQL in a moment. But bear in mind what the Data Adaptor is doing: ***Acting as a go-between for the Connection Object and the Data Set***

Structured Query Language

SQL (pronounced SeeKwel), is short for Structured Query Language, and is a way to query and write to databases (not just Access). The basics are quite easy to learn. If you want to grab all of the records from a table in a database, you use the SELECT word. Like this:

SELECT * FROM Table_Name

SQL is not case sensitive, so the above line could be written:

Select * from Table_Name

But your SQL statements are easier to read if you type the keywords in uppercase letters. The keywords in the lines above are **SELECT** and **FROM**. The asterisk means "All Records". Table_Name is the name of a table in your database. So the whole line reads:

"SELECT all the records FROM the table called Table_Name"

You don't need to select all (*) the records from your database. You can just select the columns that you need. The name of the table in our database is **tblContacts**. If we wanted to select just the first name and surname columns from this table, we can specify that in our SQL String:

SELECT tblContacts.FirstName, tblContacts.Surname FROM tblContacts

When this SQL statement is executed, only the FirstName and Surname columns from the database will be returned.

There are a lot more SQL commands, but for our purposes this is enough.

Because we want to SELECT all (*) the records from the table called tblContacts, we pass this string to the string variable we have called sql:

sql = "SELECT * FROM tblContacts"

Your code window should now look like this (though the file path to your database might be different):

```
Private Sub btnLoad_Click(ByVal sender As System.Object, _
                            ByVal e As System.EventArgs) _
                                Handles btnLoad.Click

    Dim con As New OleDb.OleDbConnection
    Dim dbProvider As String
    Dim dbSource As String
    Dim ds As New DataSet
    Dim da As OleDb.OleDbDataAdapter
    Dim sql As String

    dbProvider = "PROVIDER=Microsoft.Jet.OLEDB.4.0;"
    dbSource = "Data Source = E:/AddressBook.mdb"

    con.ConnectionString = dbProvider & dbSource

    con.Open()

        sql = "SELECT * FROM tblContacts"
        da = New OleDb.OleDbDataAdapter(Sql, con)

    MsgBox("Database is now open")

    con.Close()

    MsgBox("Database is now Closed")

End Sub
```

Now that the Data Adapter has selected all of the records from the table in our database, we need somewhere to put those records - in the **DataSet**.

Filling the DataSet

The Data Adapter can Fill a DataSet with records from a Table. You only need a single line of code to do this:

da.Fill(ds, "AddressBook")

As soon as you type the name of your Data Adapter (**da** for us), you'll get a pop up box of properties and methods. Select Fill from the list, then type a pair of round brackets. In between the round brackets, you need two things: the **Name** of your DataSet (**ds**, in our case), and an identifying name. This identifying name can be anything you like. But it is just used to identify this particular Data Adapter Fill. We could have called it "Bacon Sandwich", if we wanted:

da.Fill(ds, "Bacon Sandwich ")

The code above still works. But it's better to stick to something a little more descriptive than "Bacon Sandwich"!

Add the new line after the creation of the Data Adaptor:

```
da = New OleDb.OleDbDataAdapter(sql, con)
da.Fill(ds, "AddressBook")
```

And that's it. The DataSet (**ds**) will now be filled with the records we selected from the table called **tblContact**. There's only one slight problem - nobody can see the data yet! We'll tackle that in the next part.

Displaying the Data in the DataSet

The first part of Databases and VB .NET can be found here:

Coding your own VB .NET database projects

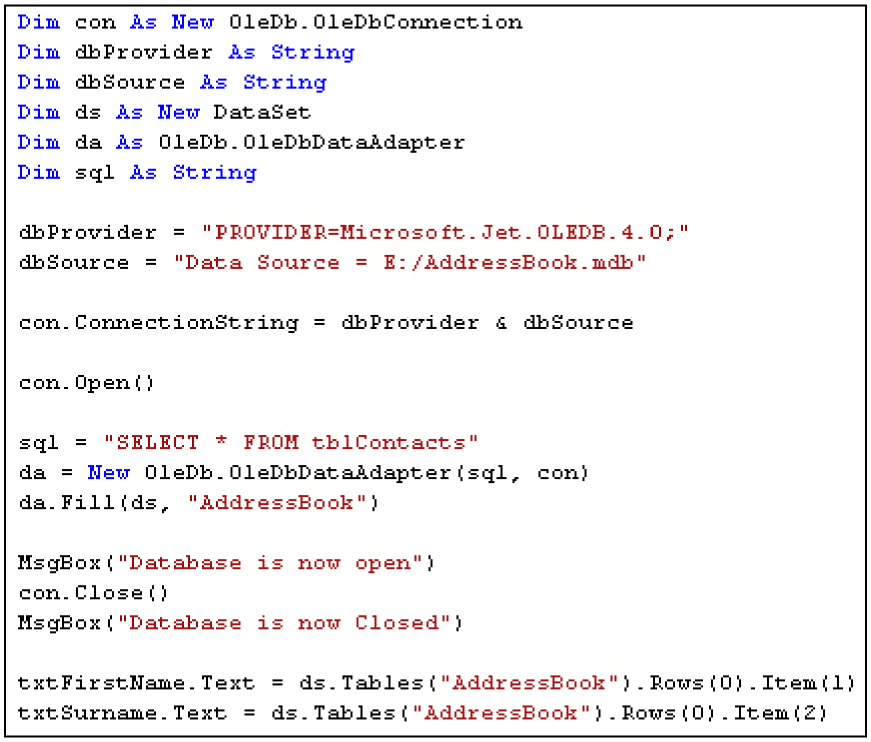
In the previous section, we saw what Data Adaptors and DataSets were. We created a Data Adaptor so that it could fill a DataSet with records from our database. What we want to do now is to display the records on a Form, so that people can see them. So so this:

- Add two textboxes to your form
- Change the **Name** properties of your textboxes to **txtFirstName** and **txtSurname**
- Go back to your code window
- Add the following two lines:

```
txtFirstName.Text = ds.Tables("AddressBook").Rows(0).Item(1)
txtSurname.Text = ds.Tables("AddressBook").Rows(0).Item(2)
```

You can add them after the line that closes the connection to the database. Once the DataSet has been filled, a connection to a database can be closed.

Your code should now look like this:



```
Dim con As New OleDb.OleDbConnection
Dim dbProvider As String
Dim dbSource As String
Dim ds As New DataSet
Dim da As OleDb.OleDbDataAdapter
Dim sql As String

dbProvider = "PROVIDER=Microsoft.Jet.OLEDB.4.0;"
dbSource = "Data Source = E:/AddressBook.mdb"

con.ConnectionString = dbProvider & dbSource

con.Open()

sql = "SELECT * FROM tblContacts"
da = New OleDb.OleDbDataAdapter(sql, con)
da.Fill(ds, "AddressBook")

MsgBox("Database is now open")
con.Close()
MsgBox("Database is now Closed")

txtFirstName.Text = ds.Tables("AddressBook").Rows(0).Item(1)
txtSurname.Text = ds.Tables("AddressBook").Rows(0).Item(2)
```

Before the code is explained, run your programme and click the button. You should see "John Smith" displayed in your two textboxes.

So let's examine the code that assigns the data from the DataSet to the textboxes. The first line was this:

txtFirstName.Text = ds.Tables("AddressBook").Rows(0).Item(1)

It's rather a long line! But after the equals sign, you type the name of your DataSet (**ds** for us). After a full stop, select **Tables** from the popup list. The **Tables** property needs something in between round brackets. Quite bizarrely, this is NOT the name of your database table! It's that identifier you used with the Data Adapter Fill. We used the identifier "**AddressBook**". If we had used "Bacon Sandwich" then we'd put this:

ds.Tables("Bacon Sandwich")

But we didn't, so our code is:

ds.Tables("AddressBook")

Type a full stop and you'll see another list popping up at you. Select **Rows** from the list. In between round brackets, you need a number. This is a Row number from the DataSet. We want the first row, which is row zero in the DataSet:

ds.Tables("AddressBook").Rows(0)

Type full stop after Rows(0) and the popup list appears again. To identify a **Column** from the DataSet, you use **Item**. In between round brackets, you type which column you want:

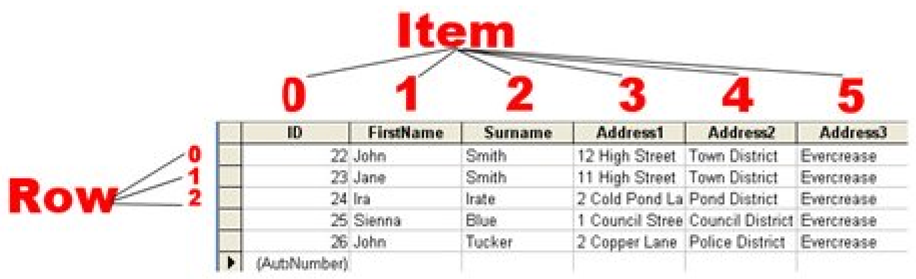
ds.Tables("AddressBook").Rows(0).Item(1)

In our Access database, column zero is used for an ID field. The **FirstName** column is the second column in our Access database. Because the Item collection is zero based, this is item 1 in the DataSet.

You can also refer to the column name itself for the Item property, rather than a number. So you can do this:

ds.Tables("AddressBook").Rows(0).Item("FirstName")
ds.Tables("AddressBook").Rows(0).Item("Surname")

If you get the name of the column wrong, then VB throws up an error. But an image might clear things up. The image below shows what the items and rows are in the database.



ID	FirstName	Surname	Address1	Address2	Address3
22	John	Smith	12 High Street	Town District	Evercrease
23	Jane	Smith	11 High Street	Town District	Evercrease
24	Ira	Irate	2 Cold Pond La	Pond District	Evercrease
25	Sienna	Blue	1 Council Stree	Council District	Evercrease
26	John	Tucker	2 Copper Lane	Police District	Evercrease
(AutoNumber)					

The image shows which are the **Rows** and which are the **Items** in the Access database Table. So the **Items** go down and the **Rows** go across.

However, we want to be able to scroll through the table. We want to be able to click a button and see the next record. Or click another button and see the previous record. You can do this by incrementing the Row number. To see the next record, we'd want this:

txtFirstName.Text = ds.Tables("AddressBook").Rows(1).Item(1)
txtSurname.Text = ds.Tables("AddressBook").Rows(1).Item(2)

The record after that would then be:

txtFirstName.Text = ds.Tables("AddressBook").Rows(2).Item(1)
txtSurname.Text = ds.Tables("AddressBook").Rows(2).Item(2)

So by incrementing and decrementing the Row number, you can navigate through the records. Let's see how that's done.

Navigate a Database with VB .NET

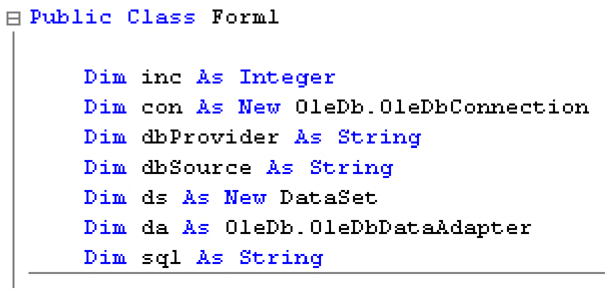
The first part of Databases and VB .NET can be found here:

Coding your own VB .NET database projects

You saw in the previous section that you can navigate through the records of a database by incrementing or decrementing the Row number of the DataSet. In this section, we're going to see a more practical example of how to do that.

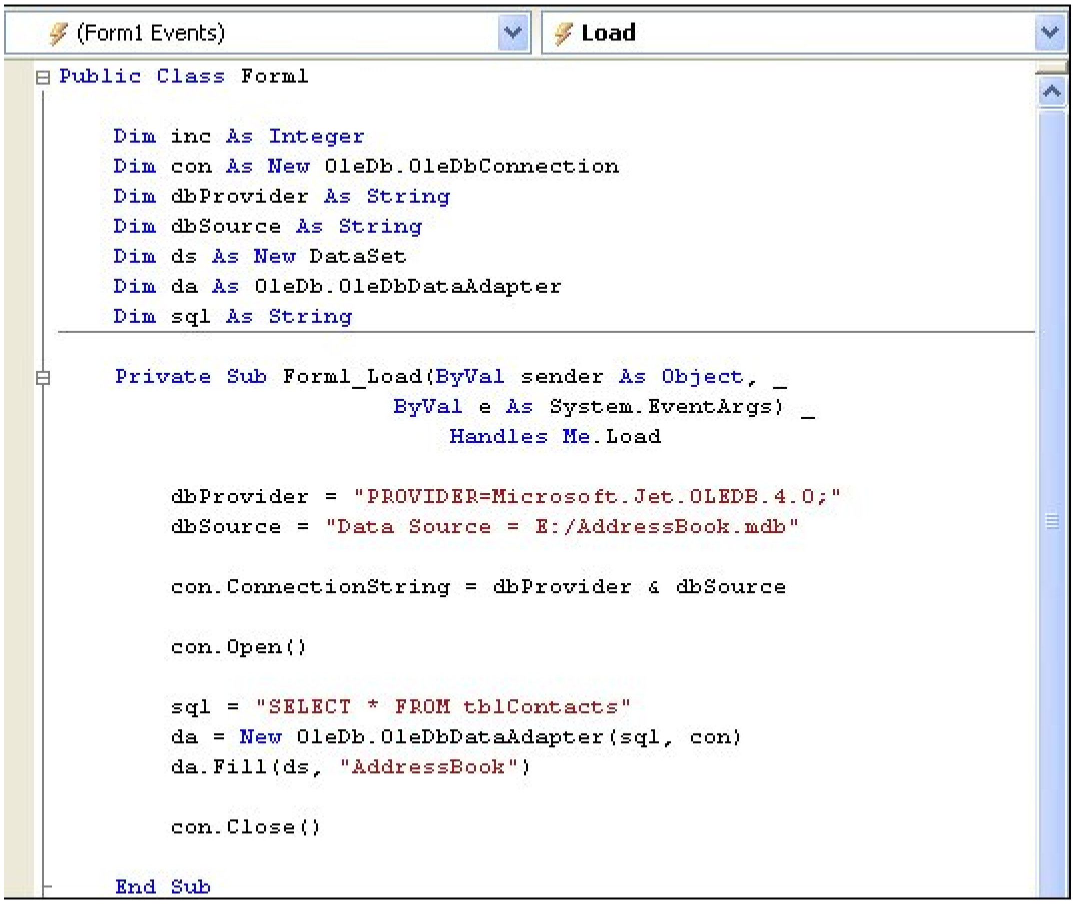
To navigate through the dataset, let's change our form. By adding some navigation buttons, we can duplicate what the wizard did. We'll also need to move the code we already have. So let's start with that.

At the moment, all our code is in the Button we added to the form. We're going to delete this button, so we need to move it out of there. The variable declarations can be moved right to the top of the coding window. That way, any button can see the variables. So move your variables declarations to the top, as in the image below (don't forget to add the **Dim inc As Integer** line):



```
Public Class Form1

    Dim inc As Integer
    Dim con As New OleDb.OleDbConnection
    Dim dbProvider As String
    Dim dbSource As String
    Dim ds As New DataSet
    Dim da As OleDb.OleDbDataAdapter
    Dim sql As String
```

We can move a few lines to the Form Load event. So, create a Form Load event, as you did in a previous section. Now move all but the textbox lines to there. Your coding window should then look like this (you can delete the message box lines, or just comment them out):



```
Public Class Form1

    Dim inc As Integer
    Dim con As New OleDb.OleDbConnection
    Dim dbProvider As String
    Dim dbSource As String
    Dim ds As New DataSet
    Dim da As OleDb.OleDbDataAdapter
    Dim sql As String

    Private Sub Form1_Load(ByVal sender As Object, _
                    ByVal e As System.EventArgs) _
                        Handles Me.Load

        dbProvider = "PROVIDER=Microsoft.Jet.OLEDB.4.0;"
        dbSource = "Data Source = E:/AddressBook.mdb"

        con.ConnectionString = dbProvider & dbSource

        con.Open()

        sql = "SELECT * FROM tblContacts"
        da = New OleDb.OleDbDataAdapter(sql, con)
        da.Fill(ds, "AddressBook")

        con.Close()

    End Sub
```

For your button, all you should have left are these two lines:

txtFirstName.Text = ds.Tables("AddressBook").Rows(inc).Item(1)
txtSurname.Text = ds.Tables("AddressBook").Rows(inc).Item(2)

Since we're going to be deleting this button, this code can be moved. Because all the buttons need to put something into the textboxes, the two lines we have left are an ideal candidate for a Subroutine. So add the following Sub to your code:

Private Sub NavigateRecords()

txtFirstName.Text = ds.Tables("AddressBook").Rows(inc).Item(1)
txtSurname.Text = ds.Tables("AddressBook").Rows(inc).Item(2)

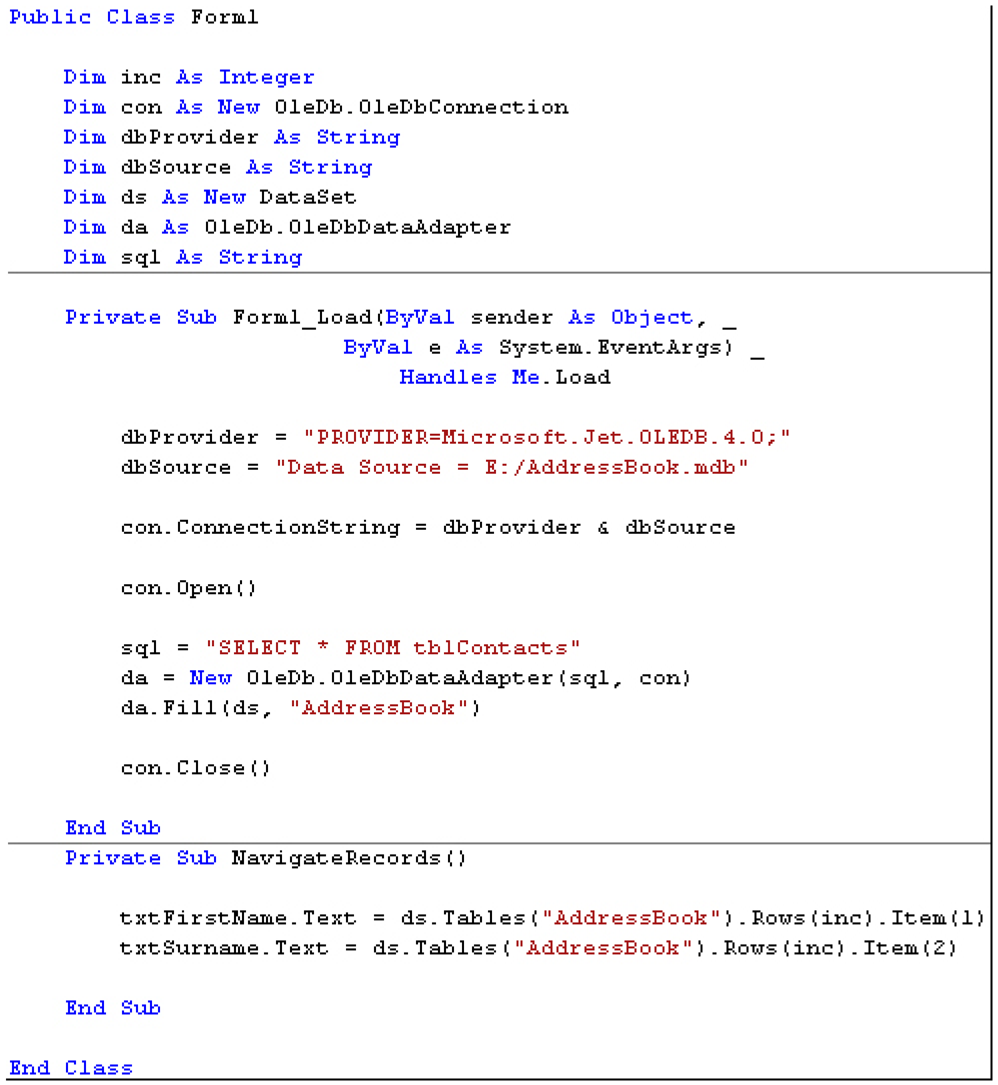
End Sub

When we navigate through the DataSet, we'll call this subroutine.

Now that all of your code has gone from your button, you can delete the button code altogether. Return to you form, click on the button to select it, then press the delete key on your keyboard.

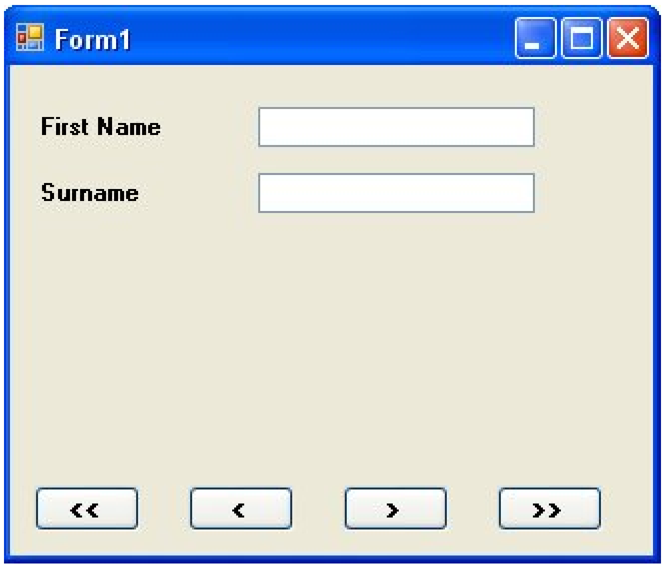
This will remove the button itself from your form. (You can also right click on the button, and then select Delete from the menu.)

Here's what your coding window should like:



```
Public Class Form1

    Dim inc As Integer
    Dim con As New OleDb.OleDbConnection
    Dim dbProvider As String
    Dim dbSource As String
    Dim ds As New DataSet
    Dim da As OleDb.OleDbDataAdapter
    Dim sql As String

    Private Sub Form1_Load(ByVal sender As Object, _
                        ByVal e As System.EventArgs) _
                            Handles Me.Load

        dbProvider = "PROVIDER=Microsoft.Jet.OLEDB.4.0;"
        dbSource = "Data Source = E:/AddressBook.mdb"

        con.ConnectionString = dbProvider & dbSource

        con.Open()

        sql = "SELECT * FROM tblContacts"
        da = New OleDb.OleDbDataAdapter(sql, con)
        da.Fill(ds, "AddressBook")

        con.Close()

    End Sub
    Private Sub NavigateRecords()

        txtFirstName.Text = ds.Tables("AddressBook").Rows(inc).Item(1)
        txtSurname.Text = ds.Tables("AddressBook").Rows(inc).Item(2)

    End Sub

End Class
```

Now you can re-design the form. Add four new buttons, and change the Name properties to: btnNext, btnPrevious, btnFirst, and btnLast. Change the Text properties to >, <, <<, and >>. Your form will then look like this:



Just a couple of more things to set up before we get started. Add a new variable declaration to the top of your code, just under the Dim inc As Integer line. Add this:

Dim MaxRows As Integer

We can store how many rows are in the DataSet with this variable. You can get how many rows are in the DataSet with this:

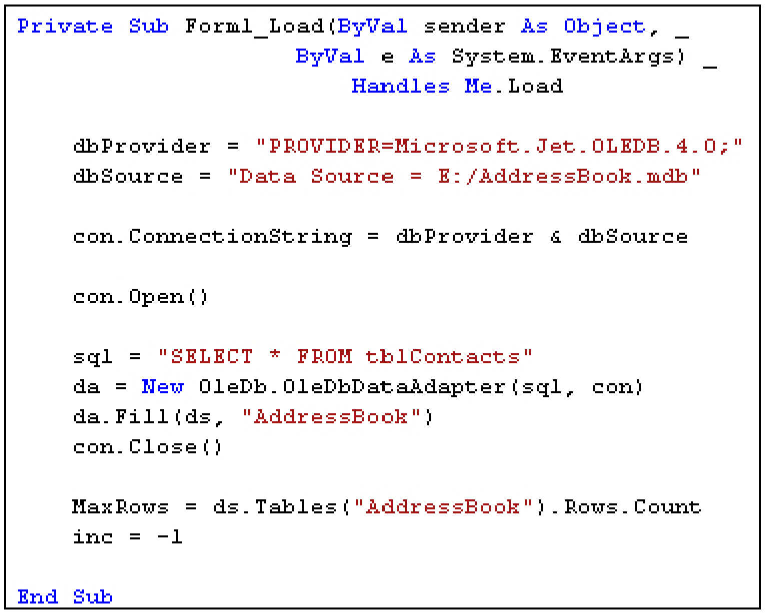
MaxRows = ds.Tables("AddressBook").Rows.Count

So the Rows property has a Count Method. This simply counts how many rows are in the DataSet. We're passing that number to a variable called MaxRows. You can then test what is in the variable, and see if the inc counter doesn't go past it. You need to do this because VB throws up an error message if try to go past the last row in the DataSet. (Previous versions of VB had some called an EOF and BOF properties. These checked the End of File and Before End of File. These properties have now gone.)

Add the following two lines of code to the Form Load Event of Form1:

MaxRows = ds.Tables("AddressBook").Rows.Count
inc = - 1

Your code should then look like this:



```
Private Sub Form1_Load(ByVal sender As Object, _
                       ByVal e As System.EventArgs) _
                           Handles Me.Load

    dbProvider = "PROVIDER=Microsoft.Jet.OLEDB.4.0;"
    dbSource = "Data Source = E:/AddressBook.mdb"

    con.ConnectionString = dbProvider & dbSource

    con.Open()

    sql = "SELECT * FROM tblContacts"
    da = New OleDb.OleDbDataAdapter(sql, con)
    da.Fill(ds, "AddressBook")
    con.Close()

    MaxRows = ds.Tables("AddressBook").Rows.Count
    inc = -1

End Sub
```

Notice the other line of code for the Form Load event:

inc = - 1

This line sets the inc variable to minus one when the form loads. When the Buttons are clicked, this will ensure that we're moving the counter on by the correct amount.

In the next Part, we'll see how the Buttons on the form work.

Coding for the Navigate Buttons

This lessons is part of an ongoing tutorial. The first part is here:

Coding your own VB .NET database projects

In the last lesson, you set up a Form with four buttons and two textboxes. You then added the following code:

In this lesson, you'll add the code for the buttons.

How to Move Forward One Record at a Time

Double click your **Next Record** button to access the code. Add the following If … Else Statement:

```
If inc <> MaxRows - 1 Then
inc = inc + 1
NavigateRecords()
Else
MsgBox("No More Rows")
End If
```

We're checking to see if the value in **inc** does not equal the value in **MaxRows** - 1. If they are both equal then we know we've reached the last record in the DataSet. In which case, we just display a message box. If they are not equal, these two lines get executed:

```
inc = inc + 1
NavigateRecords()
```

First, we move the **inc** counter on by one. Then we call the Sub we set up:

NavigateRecords()

Our Subroutine is where the action takes place, and the values from the DataSet are placed in the textboxes. Here it is again:

```
Private Sub NavigateRecords()

txtFirstName.Text = ds.Tables("AddressBook").Rows(inc).Item(1)
txtSurname.Text = ds.Tables("AddressBook").Rows(inc).Item(2)

End Sub
```

The part that moves the record forward (and backwards soon) is this part:

Rows(inc)

Previously, we hard-coded this with:

Rows(0)

Now the value is coming from the variable called **inc**. Because we're incrementing this variable with code, the value will change each time the button is clicked. And so a different record will be displayed.

You can test out your Next button. Run your programme and click the button. You should now be able to move forward through the DataSet. When you get to the end, you should see the message box display "No More Rows".

None of the other button will work yet, of course. So let's move backwards.

Move Back One Record at a Time

To move backwards through the DataSet, we need to decrement the **inc** counter. All this means is deducting 1 from whatever is currently in inc.

But we also need to check that inc doesn't go past zero, which is the first record in the DataSet. Here's the code to add to your **btnPrevious**:

```
If inc > 0 Then
inc = inc - 1
NavigateRecords()
Else
MsgBox("First Record")
End If
```

So the If statement first checks that **inc** is greater than zero. If it is, inc gets 1 deducted from. Then the NavigateRecords() subroutine gets called. If **inc** is zero or less, then we display a message.

When you've finished adding the code, test your programme out. Click the Previous button first. The message box should display, even though no records have been loaded into the textboxes. This is because the variable **inc** has a value of -1 when the form first loads. It only gets moved on to zero when the Next button is clicked. You could amend your IF Statement to this:

```
If inc > 0 Then
inc = inc - 1
NavigateRecords()
ElseIf inc = -1 Then
MsgBox("No Records Yet")
ElseIf inc = 0 Then
MsgBox("First Record")
End If
```

This new If Statement now checks to see if inc is equal to minus 1, and displays a message if it does. It also checks if inc is equal to zero, and displays the "First Record" message box.

Moving to the Last Record in the DataSet

To jump to the last record in the DataSet, you only need to know how many records have been loaded into the DataSet - the **MaxRows** variable in our code. You can then set the **inc** counter to that value, but minus 1. Here's the code to add to your **btnLast**:

```
If inc <> MaxRows - 1 Then
inc = MaxRows - 1
NavigateRecords()
End If
```

The reason we're saying **MaxRows - 1** is that the row count might be 5, say, but the first record in the DataSet starts at zero. So the total number of records would be zero to 4. Inside of the If Statement, we're setting the **inc** counter to MaxRows - 1, then calling the NavigateRecords() subroutine.

That's all we need to do. So run your programme. Click the Last button, and you should see the last record displayed in your textboxes.

Moving to the First Record in the DataSet

Moving to the first record is fairly straightforward. We only need to set the **inc** counter to zero, if it's not already at that value. Then call the Sub:

```
If inc <> 0 Then
inc = 0
NavigateRecords()
End If
```

Add the code to your **btnFirst**. Run your programme and test out all of your buttons. You should be able to move through the names in the database, and jump to the first and last records.

As yet, though, we don't have a way to add new records, to update records, or to delete them. Let's do that next.

Add, Update and Delete Records

This lessons is part of an ongoing tutorial. The first part is here:

Coding your own VB .NET database projects

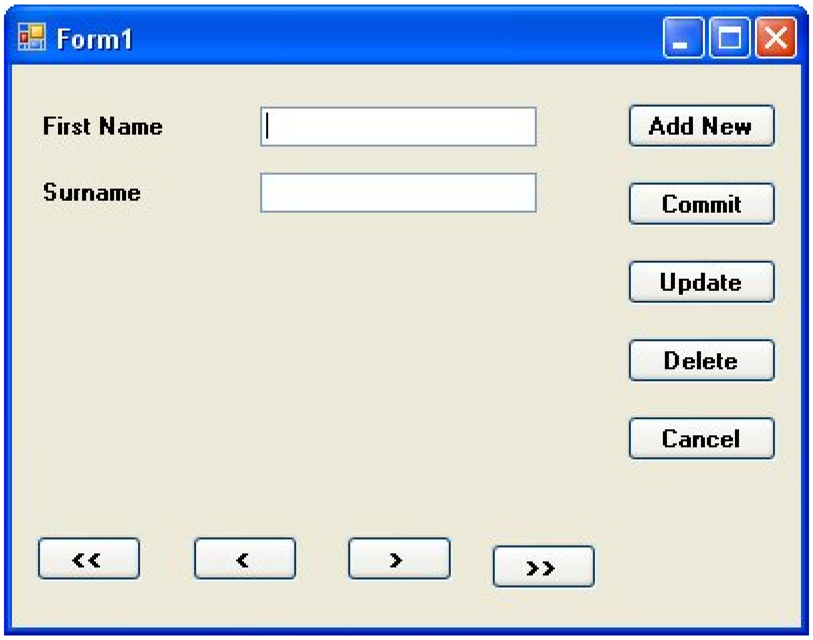
In the last section, you learned how to move through the records in your DataSet, and how to display the records in Textboxes on your form. In this lesson, we'll see how to add new records, how to delete them and how to Update a records.

Before we start the coding for these new buttons, it's important to understand that the DataSet is ***disconnected*** from the database. What this means is that if you're adding a new record, you're not adding it to the database: you're adding it to the **DataSet**! Similarly, if you're updating or Deleting, you doing it to the DataSet, and **NOT** to the database. After you have made all of your changes, you THEN commit these changes to the database. You do this by issuing a separate command. But we'll see how it all works.

You'll need to add a few more buttons to your form - five of them. Change the **Name** properties of the new Buttons to the following:

btnAddNew
btnCommit
btnUpdate
btnDelete
btnClear

Change the **Text** properties of the buttons to "**Add New Record** ", "**Commit Changes**", "**Update Record** ", "**Delete Record**", and "**Clear/Cancel**". Your form might look something like this:



We'll start with the Update Record button

Updating a Record

To reference a particular column (item) in a row of the DataSet, the code is this:

```
ds.Tables("AddressBook").Rows(2).Item(1)
```

That will return whatever is at Item 1 on Row 2.

As well as returning a value, you can also set a value. You do it like this:

```
ds.Tables("AddressBook").Rows(2).Item(1) = "Jane"
```

Now Item 1 Row 2 will contain the text "Jane". This won't, however, effect the database! The changes will just get made to the **DataSet**. To illustrate this, add the following code to your **btnUpdate**:

```
ds.Tables("AddressBook").Rows(inc).Item(1) = txtFirstName.Text
ds.Tables("AddressBook").Rows(inc).Item(2) = txtSurname.Text

MsgBox("Data updated")
```

Run your programme, and click the **Next Record** button to move to the first record. "John" should be displayed in your first textbox, and "Smith" in the second textbox. Click inside the textboxes and change "John" to "Joan" and "Smith" to "Smithy". (Without the quotes). Now click your **Update Record** button. Move to the next record by clicking your **Next Record** button, and then move back to the first record. You should see that the first record is now "Joan Smithy".

Close down your programme, then run it again. Click the **Next Record** button to move to the first record. It will still be "John Smith". The data you updated has been lost! So here, again, is why:

"Changes are made to the DataSet, and NOT to the Database"

To update the database, you need some extra code. Amend your code to this (the new lines are in bold, red text):

```
Dim cb As New OleDb.OleDbCommandBuilder(da)
```

ds.Tables("AddressBook").Rows(inc).Item(1) = txtFirstName.Text
ds.Tables("AddressBook").Rows(inc).Item(2) = txtSurname.Text

da.Update(ds, "AddressBook")

MsgBox("Data updated")

The first new line is this:

Dim cb As New OleDb.OleDbCommandBuilder(da)

To update the database itself, you need something called a **Command Builder**. The Command Builder will build a SQL string for you. In between round brackets, you type the name of your Data Adapter, **da** in our case. The command builder is then stored in a variable, which we have called **cb**.

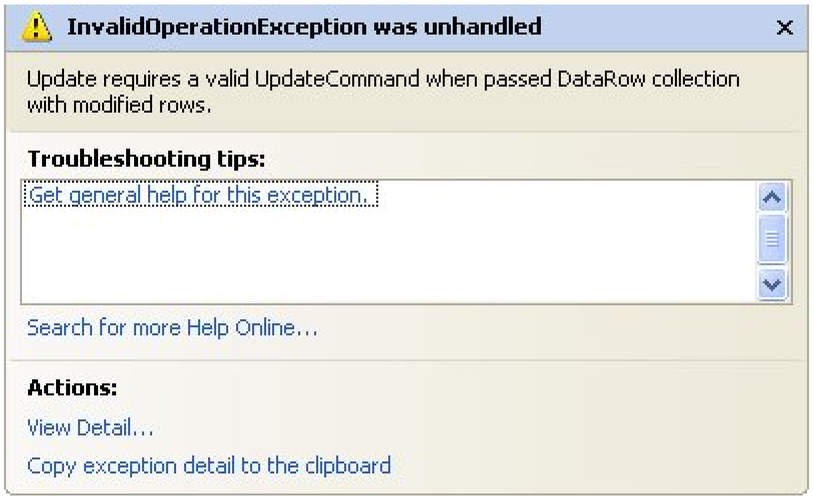
The second new line is where the action is:

da.Update(ds, "AddressBook")

The **da** variable is holding our Data Adapter. One of the methods of the Data Adapter is **Update**. In between the round brackets, you need the name of your DataSet (**ds**, for us). The "**AddressBook**" part is optional. It's what we've called our DataSet, and is here to avoid any confusion.

But the Data Adapter will then contact the database. Because we have a Command Builder, the Data Adapter can then update your database with the values from the DataSet.

Without the Command Builder, though, the Data Adapter can't do it's job. Try this. Comment out the Command Builder line (put a single quote before the "D" of Dim). Run your programme again, and then try and update a record. You'll get this error message:


InvalidOperationException was unhandled

Update requires a valid UpdateCommand when passed DataRow collection with modified rows.

Troubleshooting tips:
Get general help for this exception.
Search for more Help Online...
Actions:
View Detail...
Copy exception detail to the clipboard

The error is because you haven't got a command builder - a Valid Update Command.
Delete the comment from your Command Builder line and the error message goes away.

You should now be able to make changes to the database itself (as long as the Access database isn't Read Only).

Try it out. Run your programme, and change one of the records. Click the **Update** button. Then close the programme down, and load it up again. You should see your new changes displayed in the textboxes.

Exercise

There's one slight problem with the code above, though. Try clicking the **Update** button before clicking the **Next Record** button. What happens? Do you know why you get the error message? Write code to stop this happening

In the next part, we'll see how to add a new record.

How to Add a New Record

This lessons is part of an ongoing tutorial. The first part is here:

Coding your own VB .NET database projects

In the previous part, you learned how to Update records in the database. In the part, we'll see how to add a new record to the database using VB .NET code.

Add a New Record

Adding a new record is slightly more complex. First, you have to add a new Row to the DataSet, then commit the new Row to the Database.

But the **Add New Record** button on our form is quite simple. The only thing it does is to switch off other buttons, and clear the textboxes, ready for a new entry. Here's the code for your **Add New Record** button:

```
btnCommit.Enabled = True
btnAddNew.Enabled = False
btnUpdate.Enabled = False
btnDelete.Enabled = False

txtFirstName.Clear()
txtSurname.Clear()
```

So three buttons are switched off when the **Add New Record** button is clicked, and one is switched on. The button that gets switched on is the Commit Changes button. The Enabled property of **btnCommit** gets set to **True**. But, for this to work, you need to set it to **False** when the form loads. So return to your Form. Click **btnCommit** to select it. Then locate the **Enabled** Property in the Properties box. Set it to **False**. When the Form starts up, the button will be switched off.

The Clear/Cancel button can be used to switch it back on again. So add this code to your btnClear:

```
btnCommit.Enabled = False
btnAddNew.Enabled = True
btnUpdate.Enabled = True
btnDelete.Enabled = True

inc = 0
NavigateRecords()
```

We're switching the **Commit Changes** button off, and the other three back on. The other two lines just make sure that we display the first record again, after the Cancel button is clicked. Otherwise the textboxes will all be blank.

To add a new record to the database, we'll use the **Commit Change**s button. So double click your **btnCommit** to access its code. Add the following:

```
If inc <> -1 Then

Dim cb As New OleDb.OleDbCommandBuilder(da)
Dim dsNewRow As DataRow

dsNewRow = ds.Tables("AddressBook").NewRow()

dsNewRow.Item("FirstName") = txtFirstName.Text
dsNewRow.Item("Surname") = txtSurname.Text

ds.Tables("AddressBook").Rows.Add(dsNewRow)

da.Update(ds, "AddressBook")
```

```
MsgBox("New Record added to the Database")

btnCommit.Enabled = False
btnAddNew.Enabled = True
btnUpdate.Enabled = True
btnDelete.Enabled = True

End If
```

The code is somewhat longer than usual, but we'll go through it.

The first line is an If Statement. We're just checking that there is a valid record to add. If there's not, the **inc** variable will be on minus 1. Inside of the If Statement, we first set up a **Command Builder**, as before. The next line is this:

```
Dim dsNewRow As DataRow
```

If you want to add a new row to your DataSet, you need a **DataRow** object. This line just sets up a variable called **dsNewRow**. The type of variable is a DataRow.

To create the new DataRow object, this line comes next:

```
dsNewRow = ds.Tables("AddressBook").NewRow()
```

We're just saying, "Create a New Row object in the AddressBook DataSet, and store this in the variable called dsNewRow." As you can see, **NewRow()** is a method of **ds.Tables**. Use this method to add rows to your DataSet.

The actual values we want to store in the rows are coming from the textboxes. So we have these two lines:

```
dsNewRow.Item("FirstName") = txtFirstName.Text
dsNewRow.Item("Surname") = txtSurname.Text
```

The **dsNewRow** object we created has a Property called **Item**. This is like the Item property you used earlier. It represents a column in your DataSet. We could have said this instead:

```
dsNewRow.Item(1) = txtFirstName.Text
dsNewRow.Item(2) = txtSurname.Text
```

The **Item** property is now using the index number of the DataSet columns, rather than the names. The results is the same, though: to store new values in these properties. We're storing the text from the textboxes to our new Row.

We now only need to call the Method that actually adds the Row to the DataSet:

ds.Tables("AddressBook").Rows.Add(dsNewRow)

To add the Row, you use the **Add** method of the Rows property of the DataSet. In between the round brackets, you need the name of your DataRow (the variable **dsNewRow**, in our case).

You should know what the rest of the code does. Here's the next line:

da.Update(ds, "AddressBook")

Again, we're just using the **Update** method of the Data Adapter, just like last time. The rest of the code just displays a message box, and resets the button.

But to add a new Row to a DataSet, here's a recap on what to do:

- Create a **DataRow** variable
- Cretae an Object from this variable by using the **NewRow()** method of the DataSet **Tables** property
- Assign values to the **Items** in the new Row
- Use the **Add** method of the DataSet to add the new row

A little more complicated, but it does work! Try your programme out. Click your **Add New Record** button. The textboxes should go blank, and three of the buttons will be switched off. Enter a new First Name and Surname, and then click the **Commit Changes** button. You should see the message box telling you that a new record has been added to the database. To see the new record, close down your programme, and run it again. The new record will be there.

In the next part, you'll learn how to delete a record from the database.

Delete a Record from a Database

This lessons is part of an ongoing tutorial. The first part is here:

Coding your own VB .NET database projects

In the last part, you saw how to Add a new record to the database using VB .NET code. In this final part, you'll learn how to delete records.

Deleting Records from a Database

The code to delete a record is a little easier than last time. Double click your **btnDelete** and add the following:

```
Dim cb As New OleDb.OleDbCommandBuilder(da)

ds.Tables("AddressBook").Rows(inc).Delete()
MaxRows = MaxRows - 1

inc = 0
NavigateRecords()
da.Update(ds, "AddressBook")
```

You've met most of it before. First we set up a Command Builder. Then we have this line:

ds.Tables("AddressBook").Rows(inc).Delete()

Just as there is an **Add** method of the DataSet Rows property, so there is a **Delete** method. You don't need anything between the round brackets, this time. We've specified the Row to delete with:

Rows(inc)

The **inc** variable is setting which particular Row we're on. When the **Delete** method is called, it is this row that will be deleted.

However, it will only be deleted from the DataSet. To delete the row from the underlying database, we have this again:

da.Update(ds, "AddressBook")

The Command Builder, in conjunction with the Data Adapter, will take care of the deleting. All you need to is call the **Update** method of the Data Adapter.

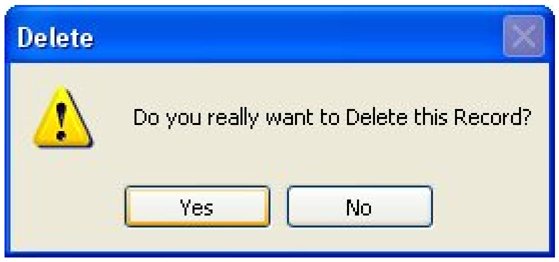
The **MaxRows** line in the code just deducts 1 from the variable. This just ensures that the number of rows in the DataSet matches the number we have in the MaxRows variable.

We also reset the **inc** variable to zero, and call the **NavigateRecords()** subroutine. This will mean that the first record is displayed, after a record has been deleted.

Try out your programme. Click the **Next Record** button a few times to move to a valid record. Then click the **Delete Record** button. The record will be deleted from the DataSet AND the database. The record that is then displayed will be the first one.

There's another problem, though: if you click the **Delete Record** button before the **Next Record** button, you'll get an error message. You can add an If Statement to check that the inc variable does not equal minus 1.

Another thing you can do is to display a message box asking users if they really want to delete this record. Here's one in action:



Do you really want to Delete this Record?

To get this in your own programme, add the following code to the very top of your Delete button code:

```
If MessageBox.Show("Do you really want to Delete this Record?", _
"Delete", MessageBoxButtons.YesNo, _
MessageBoxIcon.Warning) = DialogResult.No Then


MsgBox("Operation Cancelled")
Exit Sub

End If
```

The first three lines of the code are really one line. The underscore has been used to spread it out, so as to fit on this page.

No more reading these lessons online - get the eBook here!

But we're using the new message box function:

MessageBox.Show()

In between the round brackets, we specifying the message to display, followed by a caption for the message box. We then have this:

MessageBoxButtons.YesNo

You won't have to type all that out; you'll be able to select it from a popup list. But what it does is give you Yes and No buttons on your message box.

After typing a comma, we selected the **MessageBoxIcon**.Warning icon from the popup list.

But you need to check which button the user clicked. This is done with this:

= DialogResult.No

Again, you select from a popup list. We want to check if the user clicked the No button. This will mean a change of mind from the user. A value of No will then be returned, which is what we're checking for in the If Statement.

The code for the If Statement itself is this:

```
MsgBox("Operation Cancelled")
Exit Sub
```

This will display another message for the user. But most importantly, the subroutine will be exited: we don't want the rest of the Delete code to be executed, if the user clicked the No button.

And that's it for our introduction to database programming. You not only saw how to construct a database programme using the Wizard, but how to write code to do this yourself. There is an awful lot more to database programming, and we've just scratched the surface. But in a beginner's course, that's all we have time for.

The section that follows is all about Forms.

Anchor and Dock Controls on a Form

n this section of the course, we'll take a look at some of the extra things you can do with VB.NET forms. First, we'll take a look at the Anchor and Dock properties of a form.

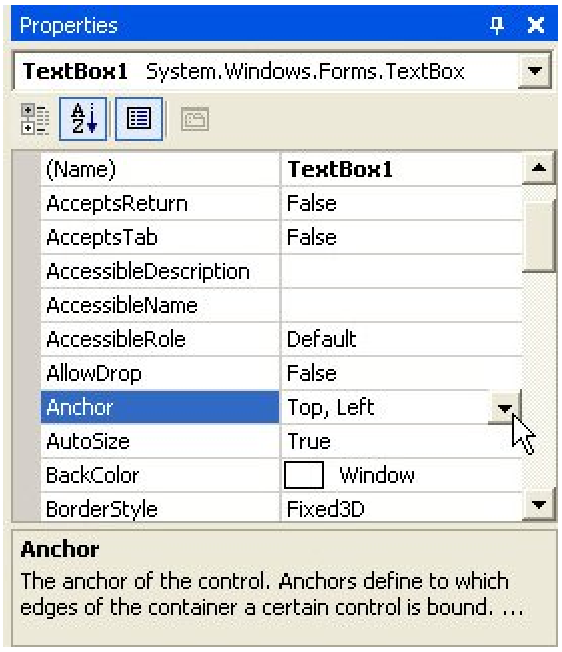
Anchoring and Docking

The Anchor and Dock properties of a form are two separate properties. Anchor refers to the position a control has relative to the edges of the form. A textbox, for example, that is anchored to the left edge of a form will stay in the same position as the form is resized. Docking refers to how much space you want the control to take up on the form. If you dock a control to the left of the form, it will stretch itself to the height of the form, but its width will stay the same. Let's take a look at some examples, to clear things up.

Anchoring

Start a new windows projects. Add two textboxes to your form, and set the MultiLine properties of both to True. Change the height of the boxes.

Click on Textbox1 and locate the Anchor property in the Properties box:

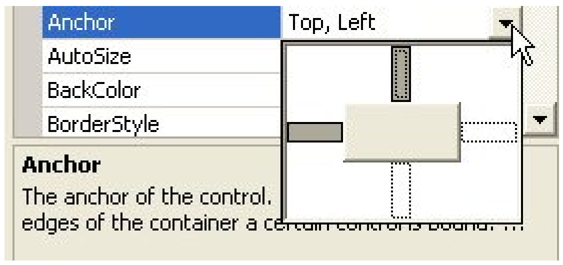


TextBox1 System.Windows.Forms.TextBox

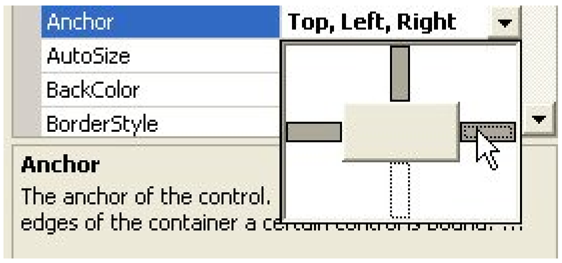
(Name)	TextBox1
AcceptsReturn	False
AcceptsTab	False
AccessibleDescription	
AccessibleName	
AccessibleRole	Default
AllowDrop	False
Anchor	Top, Left
AutoSize	True
BackColor	Window
BorderStyle	Fixed3D

Anchor
The anchor of the control. Anchors define to which edges of the container a certain control is bound. ...

The default is to anchor the control to the Top, Left edge of the form. Click the arrow to reveal a curious drop down box:

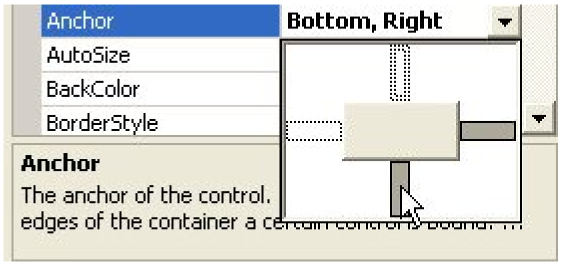


The button in the middle represents your control. The big white areas are rather confusing - they don't actually do anything! To change the property, you click the smaller grey or white rectangles between the big white rectangle. Click again to deselect it. In the image below, the property has been changed so that the textbox is anchored to the Top, Left and Right sides of the form:


Anchor: Top, Left, Right

AutoSize
BackColor
BorderStyle
Anchor
The anchor of the control.
edges of the container a certain controls bound. ...

The next image has the textbox anchored to the Right and Bottom edges of the Form:


Anchor: Bottom, Right

AutoSize
BackColor
BorderStyle
Anchor
The anchor of the control.
edges of the container a certain controls bound. ...

Notice where the cursor is in the images, and what has been changed. Click the arrow on the drop down box to confirm your choices.

To see what effect this all has, do the following:

- Set the Anchor property of Textbox1 on the default of **Top**, **Left**
- Change the Anchor property of Textbox2 to **None** (all the small rectangles should be white.)
- Run your programme and drag the edges of the Form outward. This will resize your form

What you should notice is that Textbox1 stays where it is, and that the left edge of Textbox2 moves.

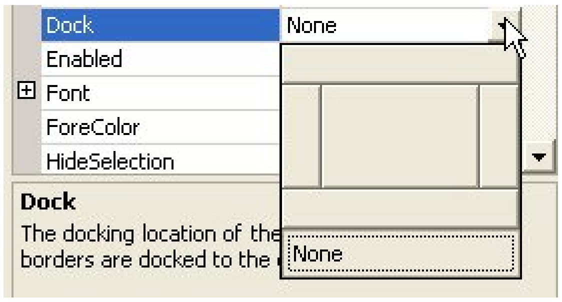
Stop your programme from running. Change the Anchor properties of the two textboxes to anything you like. Run your form again and watch what happens. Try anchoring one textbox to the left and right of the form. Watch what happens.

But anchoring a control to an edge of the form is a useful property to get used to, if you have a form that can be resized and want your controls to stay where they are.

No more reading these lessons online - get the eBook here!

Docking

Docking is similar to Anchoring, but this time the control fills a certain area of the form. To see how it works, click on one of your textboxes and locate the Dock property. Click the arrow to reveal a drop down box:



This time, all the rectangles are like buttons. You can only dock to one side at a time, and the default is None. Click a button to see what it does to your textbox. Click the middle one, and the textbox will Fill the whole form.

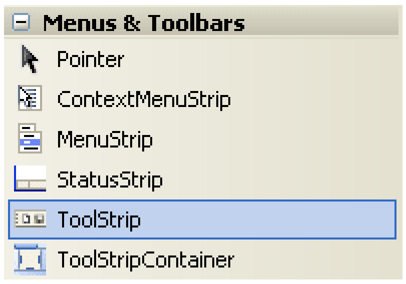
Docking is quite useful when used with the splitter control and panels, allowing you to create a Windows-style interface.

In the next part, we'll take a look at how to add a Toolbar to your Form.

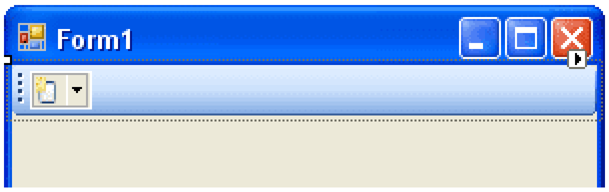
Adding a Toolbar to a Form

The toolbar is a very popular and much-used addition to a programme. It's difficult to think of a piece of software that doesn't make use of them. VB.NET lets you add toolbars to your forms, and the process is quite straightforward. Let's see how it's done:

Either start a new Windows project, or keep the one you currently have. To add a toolbar to the top of your form, expand the Toolbox and locate the ToolStrip control:



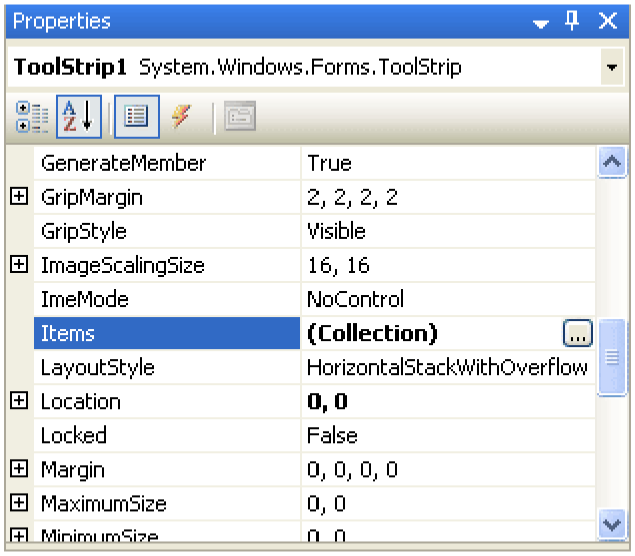
Double click the ToolStrip control, and it will be added to the top of your form:



You should also notice the ToolStrip object that appears at the bottom of the window:

ToolStrips work by adding buttons and images to them. The button is then clicked, and an action performed.

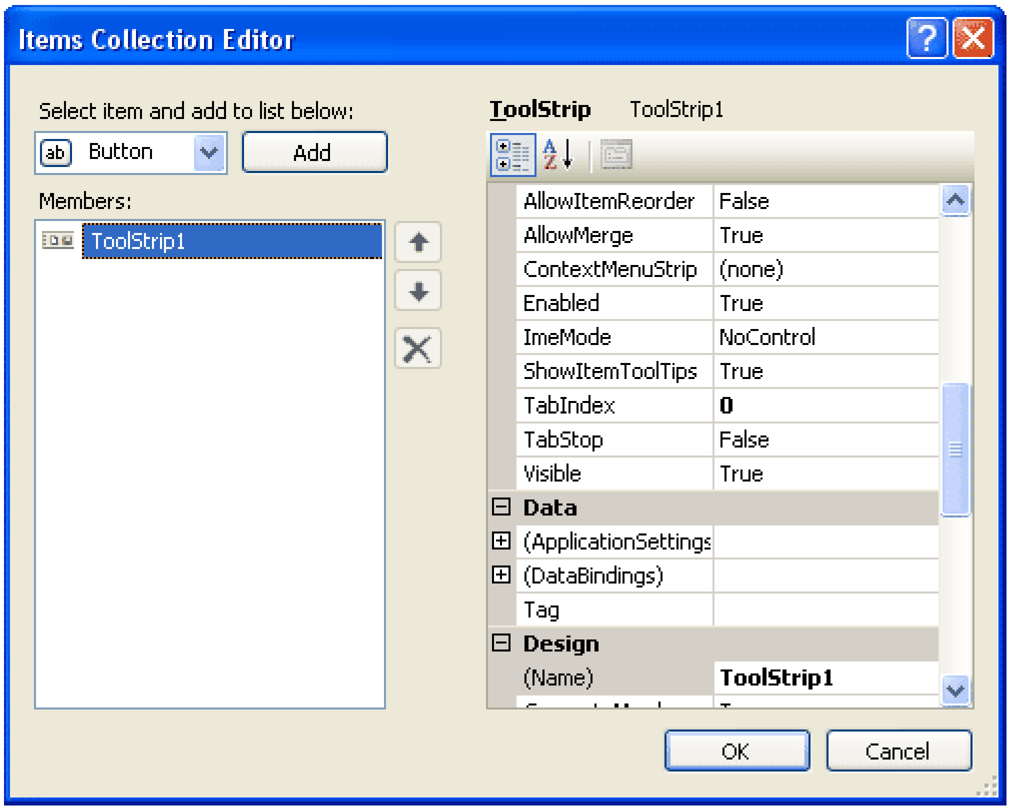
Click on your ToolStrip to select it. In the property box for the ToolStrip, you'll notice that it has the default Name of **ToolStrip1**. We'll keep this Name. But locate the Items (Collection) property:



ToolStrip1 System.Windows.Forms.ToolStrip

GenerateMember	True
GripMargin	2, 2, 2, 2
GripStyle	Visible
ImageScalingSize	16, 16
ImeMode	NoControl
Items	**(Collection)**
LayoutStyle	HorizontalStackWithOverflow
Location	**0, 0**
Locked	False
Margin	0, 0, 0, 0
MaximumSize	0, 0
MinimumSize	0, 0

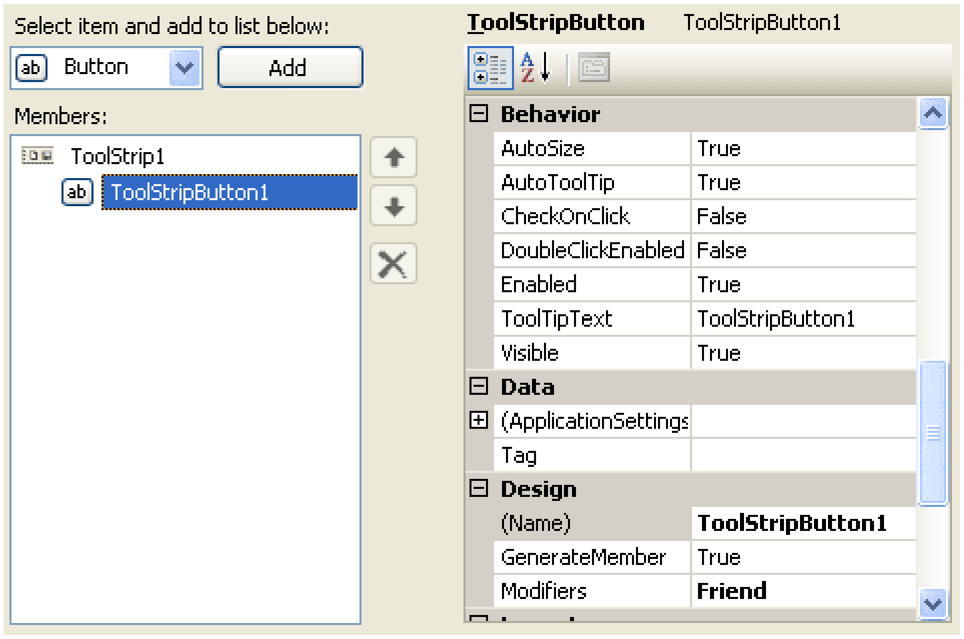
Click the button with the three dots in it. This brings up the Items Collection Editor:



ToolStrip ToolStrip1

AllowItemReorder	False
AllowMerge	True
ContextMenuStrip	(none)
Enabled	True
ImeMode	NoControl
ShowItemToolTips	True
TabIndex	**0**
TabStop	False
Visible	True
Data	
(ApplicationSettings	
(DataBindings)	
Tag	
Design	
(Name)	**ToolStrip1**

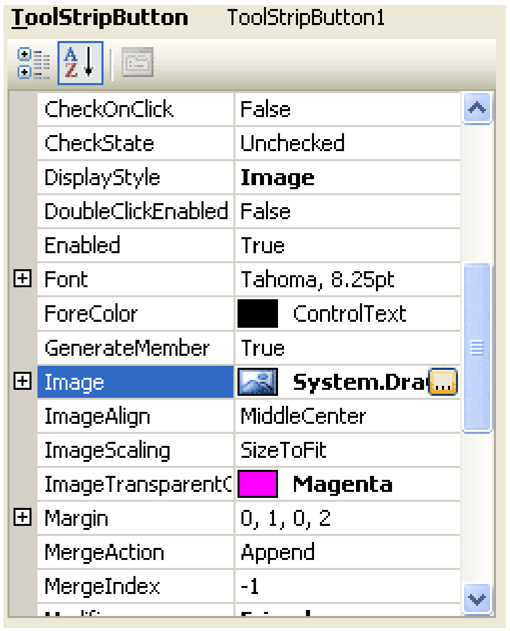
To add a new button to your ToolStrip, click the **Add** button at the top. The button appears in the Members box (ToolStripButton1):



ToolStripButton ToolStripButton1

Behavior	
AutoSize	True
AutoToolTip	True
CheckOnClick	False
DoubleClickEnabled	False
Enabled	True
ToolTipText	ToolStripButton1
Visible	True
Data	
(ApplicationSettings	
Tag	
Design	
(Name)	**ToolStripButton1**
GenerateMember	True
Modifiers	**Friend**

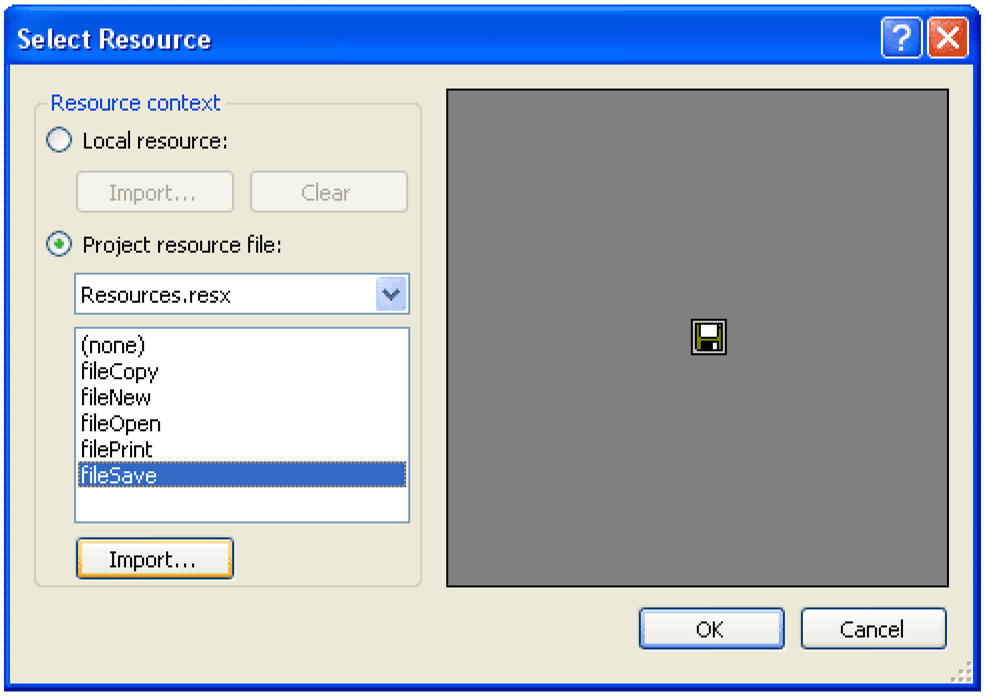
Notice that the new button has its own list of properties, just to the right. To add an image to this new button, locate the Image property:



ToolStripButton ToolStripButton1

CheckOnClick	False
CheckState	Unchecked
DisplayStyle	**Image**
DoubleClickEnabled	False
Enabled	True
Font	Tahoma, 8.25pt
ForeColor	■ ControlText
GenerateMember	True
Image	**System.Dra**
ImageAlign	MiddleCenter
ImageScaling	SizeToFit
ImageTransparent(	🟪 **Magenta**
Margin	0, 1, 0, 2
MergeAction	Append
MergeIndex	-1

Click the small button with the 3 dots in it to bring up the Select Resource box:



Select Resource

Resource context

○ Local resource:

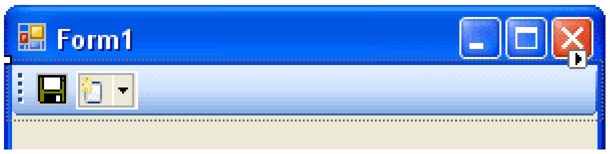
Import... Clear

● Project resource file:

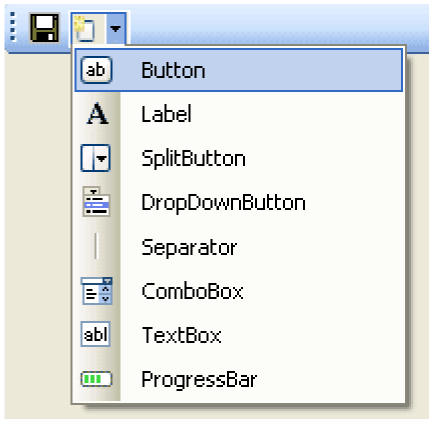
Resources.resx

(none)
fileCopy
fileNew
fileOpen
filePrint
fileSave

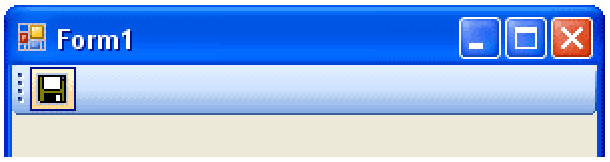
In the image above, we've selected "Project resource file", and then clicked the Import button. We then navigated to some Bitmap images and imported the five that you can see in the screenshot above. (The Bitmap folder is amongst the files you download at the start of this book.) Click OK when you have imported some images. You will be returned to the Item Collection Editor. Click OK on this, as well. The ToolStrip on your form will then look like this:



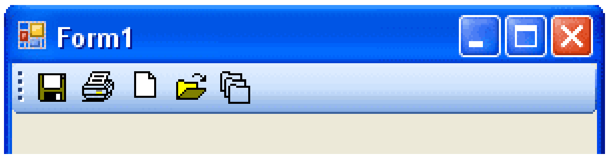
The second of those images is a dropdown list of available ToolStrip options:



So if you want, say, a separator instead of a button, select it from the list. This dropdown list will disappear when you run the form:



Repeat the steps outlined above, and add some more buttons to your ToolStrip. It should then look something like ours:



If your ToolStrip images have a grey background, you can get rid of this by bringing up the Image Collection Editor again. Select a button, and then locate the property called ImageTransparentColor. The default is Magenta. Change it to Silver, which is one of the Custom colours (second grey down, on the left).

Of course, if you click on the buttons nothing will happen. We need to write the code that gets them to work.

Coding For your Toolbar buttons

Double click your first ToolStripbutton to bring up the coding window. It should look like this:

```
Private Sub ToolStripButton1_Click(ByVal sender As System.Object, _
                                   ByVal e As System.EventArgs) _
                                       Handles ToolStripButton1.Click

End Sub
```

You can place any code you like, here. Try a message box, as in the image below:

```
Private Sub ToolStripButton1_Click(ByVal sender As System.Object, _
                                   ByVal e As System.EventArgs) _
                                       Handles ToolStripButton1.Click

    MessageBox.Show("Save button clicked")

End Sub
```

Run your programme and click your ToolStrip button. You should see the message box display. In a real programme, however, the code would be the same code for a menu item - it's just shortcut, after all!

In the next part, you'll learn how to create multiple forms in a VB NET project.

Creating Multiple Forms in VB .NET

It's a rare programme that only has one form in it. Most programmes will have other forms. These other forms can be used for things like Find and Replace searches, extra formatting capabilities, to set Options for the programme, and a whole lot more besides. VB.NET let's you add as many forms as you want to your project. But the process is not quite so simple. We'll see how to do it, though.

You can use the form you already have for this, the one with the ToolStrip on it (or start a new project, if you prefer). But from the VB.NET design environment, click the **Project** menu. From the drop down menu, click **Add Windows Form**. The Add New Item dialogue box appears.

Select **Windows Form** under Templates. Then click inside the Name textbox at the bottom. Change the Name of the form to **frmSecond.vb**. Then click Add.

When you are returned to the design environment, your new form will be displayed:

To switch between forms, you can click the tabs. In the image, two tabs are displayed: Form1 (the original and first form), and our new form **frmSecond**.

We'll write code to get this new form to display. But it will only appear when a button is clicked on Form1.

So click the tab for Form1, and add a button to this form. Change the **Name** property of the button to **btnShowSecond**. Then double click the button to access the code for it.

In order to display the second form, you have to bear in mind that Forms are Classes. So **frmSecond** is a Class (as is Form1). You first have to create a new object from the class called frmSecond Class. Then call its Show method.

So add this code to your button

```
Dim SecondForm As New frmSecond

SecondForm.Show()
```

The first line sets up a variable called **SecondForm**. When you type "**As New**", you're asking VB.NET to create a New object. If you type a space, you'll see a pop up list. Type the **frm** of frmSecond and you should see it displayed on the list. You can double click the item in the list to add it to your code. But what the line does is create a new Object from the Class called **frmSecond**.

Once we have the Form Object stored in the variable, we can just use the Show method to display the form.

Run your programme and test it out. When you click your button, you should see the second form appear.

However, there's a problem with this code. Click the button again and another copy of **frmSecond** appears. Keep clicking the button and your screen will be filled with the second form!

To prevent this from happening, you can move the code that creates the form object. Move it right to the top of the coding window, just below **Public Class Form1**.

The only code left in the button is the line that Shows the form. A new form object will now not be created every time the button is clicked. If you try it out, you should see only one form appear when the button is clicked, and not multiple forms.

In the next part, we'll take a look at Modal and Non Modal forms.

Modal and Non Modal Forms

This lesson follows on from the previous short lesson: How to create a second form

A modal from is one that has to be dealt with before a user can continue. An example is the Change Case dialogue box in Microsoft Word. If you try to click away from the dialogue box, you'll here a beep to indicate an error. Until you click either the Cancel or OK buttons, the programme won't let you click anywhere else.

The second form you've just created is called a Modeless form. These are forms than can be hidden or sent to the taskbar. You can then return to the main form or programme and do things with it.

A Modal form is sometimes called a dialogue box. And we'll see how to create one of these now.

Add a second button to your Form1. Change the **Name** property of the new button to **btnDialogueBox**. Double click the new button and add the following code:

```
Dim frmDialogue As New frmSecond

frmDialogue.ShowDialog()
```

To display a form as a Modal dialogue box, you use the **ShowDialog** method. If you use the Show method, the form is displayed as a Modeless form.

Run your programme. Click your new button, and the second form should display. Move it out the way and try to click a button on Form1. You won't be able to. The second form has to be dealt with before you can access Form1.

When the form is a Modal dialogue box, you can create OK and Cancel buttons for it. VB.NET then has a trick up its sleeve for these types of buttons. We'll see that trick now.

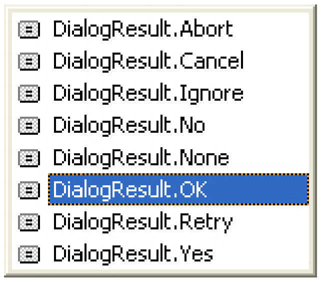
No more reading these lessons online - get the eBook here!

OK and Cancel Buttons

In the design environment, Click the Tab for your **frmSecond**. When the form is displayed in the design window, add two buttons to it (Make sure you're adding the buttons to the second form and NOT Form1). Change the **Name** property of the first button to **btnOK**, and the **Name** property of the second to **btnCancel**. Double click your OK button and add the following code to it:

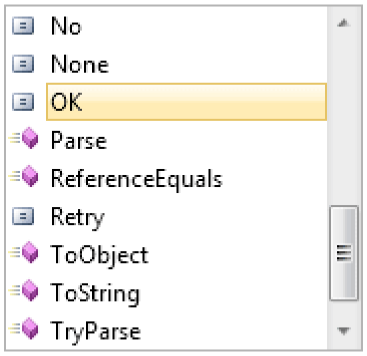
Me.DialogResult = DialogResult.OK

The **Me** keyword refers to the current form. When you type a full stop, select **DialogResult** from the pop up list that appears. DialogResult is a property of the Form. It can accept a range of values. As soon as you type a space after the equals sign, you'll see a list with these values on it (VB NET 2008 only. In VB 2010, you have to type the DialogResult):



As you can see, in VB NET 2008, one of these values is **DialogResult.OK**. This indicates that you want to use this button as an OK button. When the button is clicked, VB.NET will return a result of OK for this button.

In VB NET 2010, type **DialogResult** after the equals sign. Type a dot and you'll have this instead of the above image:



Access the code for your Cancel button and add the following line:

Me.DialogResult = DialogResult.Cancel

For the Cancel button, we're just selecting **DialogResult.Cancel** from the list. When the button is clicked, VB.NET will return a result of Cancel for this button.

You can test to see what value is stored in **Me.DialogResult**. But you do that from the button that displays the form, **Form1** for us.

So access your Form1 code, and locate the lines that display the second form. The two lines should be these:

```
Dim frmDialogue As New frmSecond

frmDialogue.ShowDialog()
```

Change the second line to this:

```
If frmDialogue.ShowDialog() = DialogResult.OK Then
MsgBox("OK Button Clicked")
End If
```

To get at the value of the button clicked, you test to see what result the **ShowDialog** property is. If the **ShowDialog** property of **frmDialogue** is **OK** then you can execute the code that needs executing. If the Cancel button was clicked, however, you don't have to do anything: VB.NET will take of closing your Modal dialogue box for you!

Run your programme and test it out. Click your button to bring up your Modal dialogue box. Click the OK button, and you should see the message box display. Bring the Modal dialogue box up a second time and then click the Cancel button. The form will just close down.

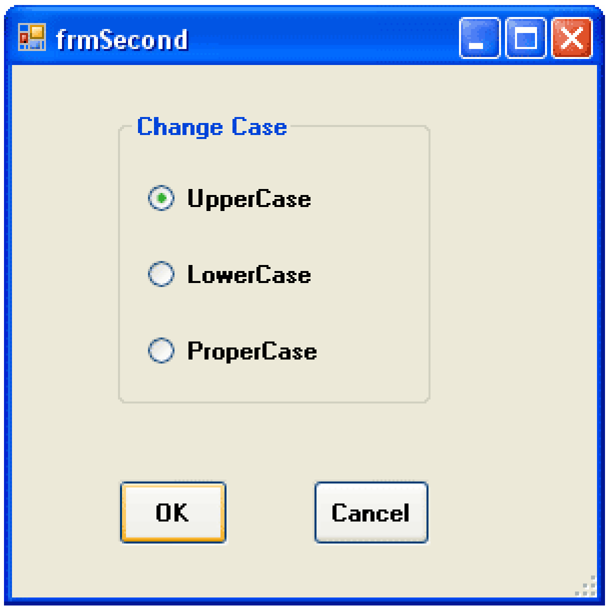
In the next part, we'll see how to return values from a second form.

Getting at Values on other Forms

This lesson follows on from the previous two lessons: How to create a second form

The form with OK and Cancel buttons on it is not doing much good. We need it do some work for us. Let's turn the form into a Change Case dialogue box.

Design a Form like the one in the following image (this is **frmSecond**):



When you've designed your form, click back on Form1 and add a Textbox to it. When the button on Form1 is clicked, the dialogue box above will display. You can then select an option button to change the case to Upper, Lower or Proper case. This will happen when the OK button is clicked. Whatever text is in Texbox1 on Form1 will be changed accordingly.

Double click the OK button on **frmSecond** to access the code. You should have the following:

```
Me.DialogResult = DialogResult.OK
```

If you want to refer to Texbox1 on Form1, you can't just do this:

```
Form1.Textbox1.Text
```

In previous version of VB, that code would be all right. You're saying "Access the Text property of Textbox1 on Form1." The problem in VB.NET is that forms are Classes. They don't become objects until one is created from a Class. So the frmSecond Class knows nothing about Form1. It has no idea what it is.

The solution is to create a textbox object variable on Form1, and assign Textbox1 to this variable. But this variable has to be something that all Classes in the project can see.

So add this near the top of your code window for Form1 (add it just below the Inherits System.Windows.Forms.Form line, or Public Class Form1):

```
Public Shared tb As TextBox
```

We're setting up a variable which we've called **tb**. A Textbox object is going to be stored in this variable. But notice that the variable is **Public Shared**. This way, **frmSecond** will be able to see the variable.

In the Form Load event for Form1, add the following line:

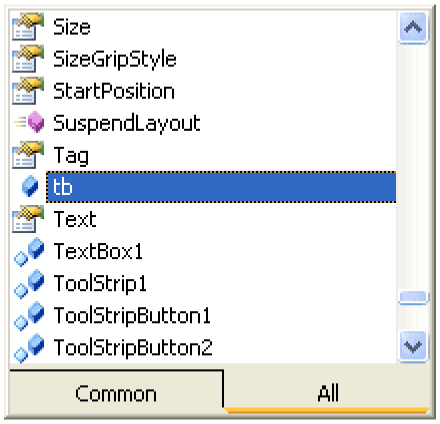
tb = Textbox1

When Form1 loads, the textbox called Textbox1 will be assigned to the tb variable. Now Textbox1 can be seen by frmSecond.

Go back to your code for the OK button on frmSecond. Add the following two lines at the top:

```
Dim ChangeCase As String
ChangeCase = Form1.tb.Text
```

We're setting up a String variable called ChangeCase. Whatever text is in Textbox1 of Form1 will then be assigned to the ChangeCase variable. But notice that as soon as you type a full stop after Form1, the **tb** variable will be available in the pop up list:



The Public variable called tb holds a reference to Textbox1 on Form1. When you type a full stop after the tb, you get a list popping up. The list is all the Properties and Methods that are available to Textbox1. One of these is the Text property.

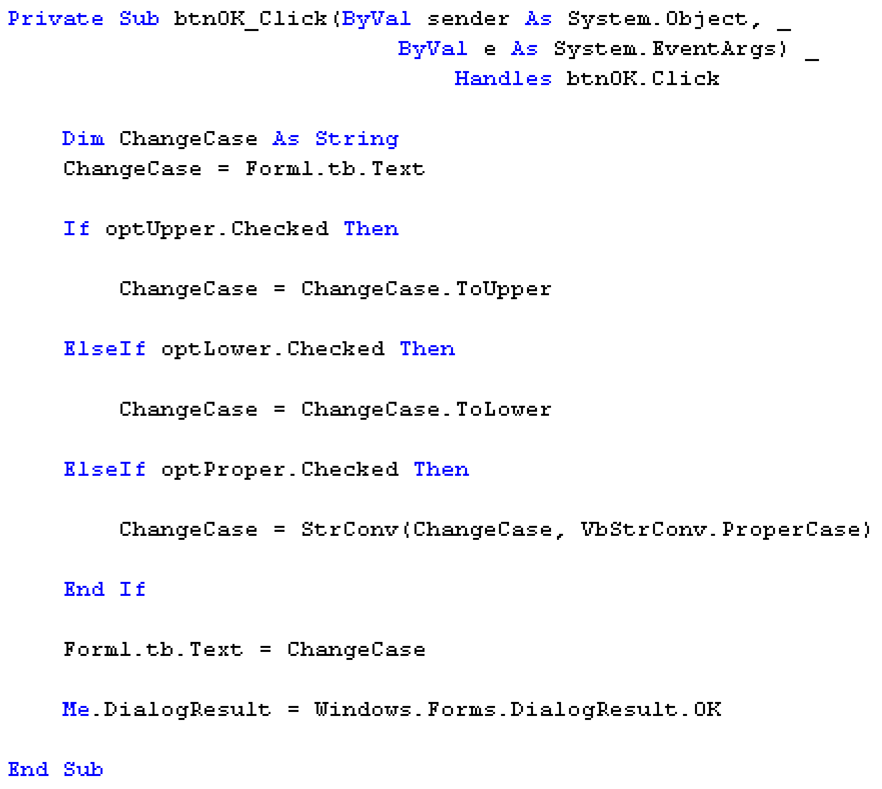
No more reading these lessons online - get the eBook here!

We now only need to add the code that does the actual converting. So add this below the two lines you already have:

```
Dim ChangeCase As String
ChangeCase = Form1.tb.Text

If optUpper.Checked Then
ChangeCase = ChangeCase.ToUpper
```

```
ElseIf optLower.Checked Then
ChangeCase = ChangeCase.ToLower
ElseIf optProper.Checked Then
ChangeCase = StrConv(ChangeCase, VbStrConv.ProperCase)
End If

Form1.tb.Text = ChangeCase
```

The three options buttons on our form were called **optUpper**, **optLower** and **optProper**. In the code, we're using an If Statement to see which of these was selected. The one that was chosen will have its **Checked** property set to **True**. We then store into the variable **ChangeCase** the converted text from the textbox. The final line puts the converted text back into Textbox1 on Form1. But you're coding window should look like this:



```
Private Sub btnOK_Click(ByVal sender As System.Object, _
                        ByVal e As System.EventArgs) _
                          Handles btnOK.Click

    Dim ChangeCase As String
    ChangeCase = Form1.tb.Text

    If optUpper.Checked Then

        ChangeCase = ChangeCase.ToUpper

    ElseIf optLower.Checked Then

        ChangeCase = ChangeCase.ToLower

    ElseIf optProper.Checked Then

        ChangeCase = StrConv(ChangeCase, VbStrConv.ProperCase)

    End If

    Form1.tb.Text = ChangeCase

    Me.DialogResult = Windows.Forms.DialogResult.OK

End Sub
```

Note that the DialogResult.OK line is the final line of the code. When you're writing your code, make sure that optUpper, optLower and optProper are changed to whatever you called your Radio Buttons.

When you're finished adding the code, run your programme. Enter some text into Textbox1. Then click the button that brings up the Change Case Dialogue box. Select an option from the three available, and the click OK. The text in Textbox1 should be converted.

Setting and Getting value from one form to another can be quite a tricky process at first. But once you get the hang of it you'll find it's not too difficult.

And that ends this section of this course. There's an awful lot more to learn about Windows Forms, and a bit of experimentation is needed before you become skilled in their use. But in a beginners course, you've learned enough to be going on with.

Section Three Exercises

Part 1 - If statements

Start a new project. Add a textbox, a Label and a button to your new Form. Then write a programme that does the following:

1. Asks users to enter a number between 10 and 20.
2. The number will be entered into the Textbox.
3. When the Button is clicked, your Visual Basic code will check the number entered in the Textbox.
4. If it is between 10 and 20, then a message will be displayed.
5. The message box will display the number from the Textbox.
6. If the number entered is not between 10 and 20 then the user will be invited to try again, and whatever was entered in the Textbox will be erased

Part 2 - Select Case Statements

Add a Combo box and another button to your form. Create a list of items for your Combo Box. The list of items in your Combo box can be anything you like - pop groups, football teams, favourite foods, anything of your choice. Then try the following:

Use a select case statement to test what a user has chosen from your drop-down list. Give the user a suitable message when the button was clicked.

In the next section of this course, we'll move on to loops in Visual Basic .NET.