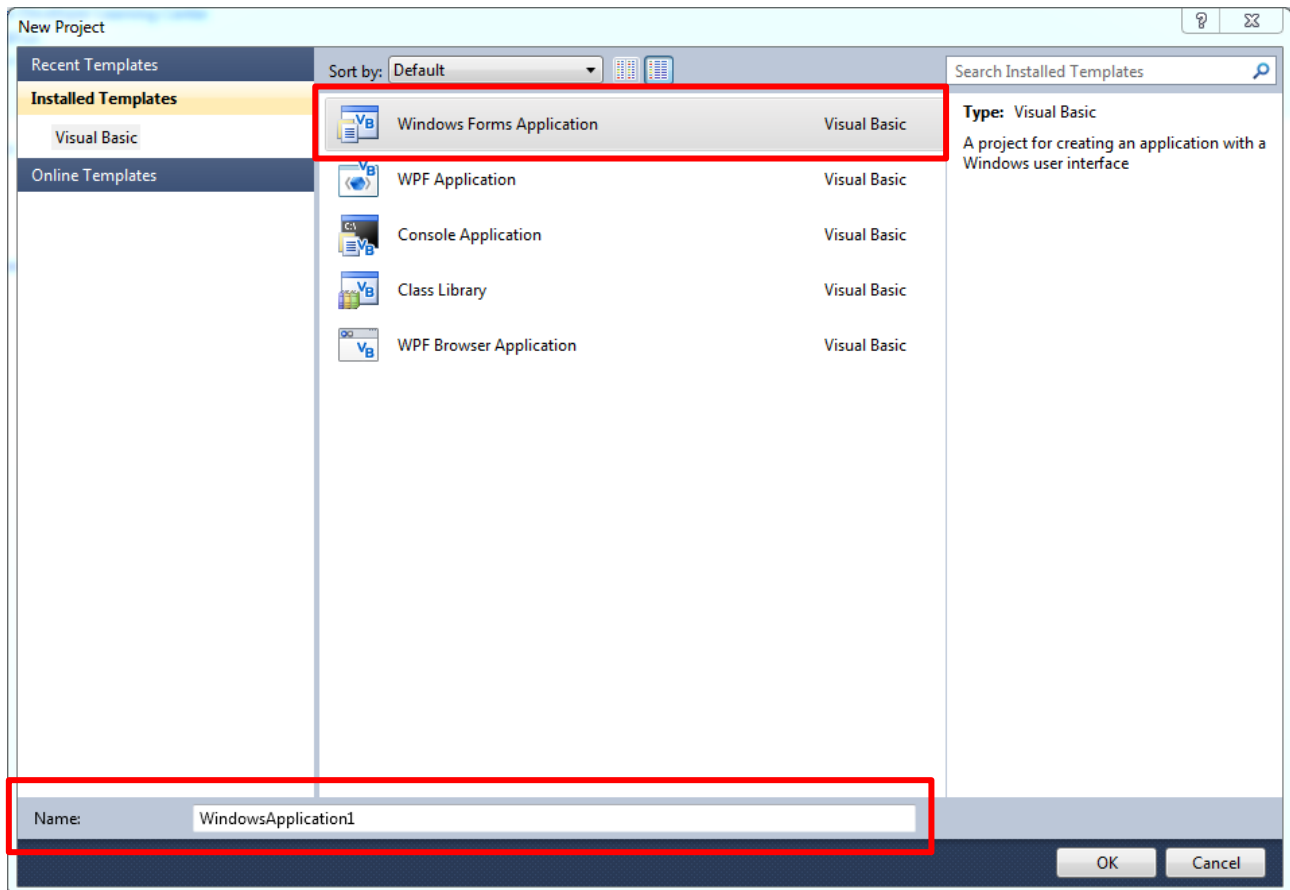


Swift to Visual Basic.NET – Transition Guide

This sheet is designed to help you manage the transition between Swift, used for Year 10 IST, and Visual Basic.NET, used for HSC Software Design and Development and IB Computer Science.

Creating a project

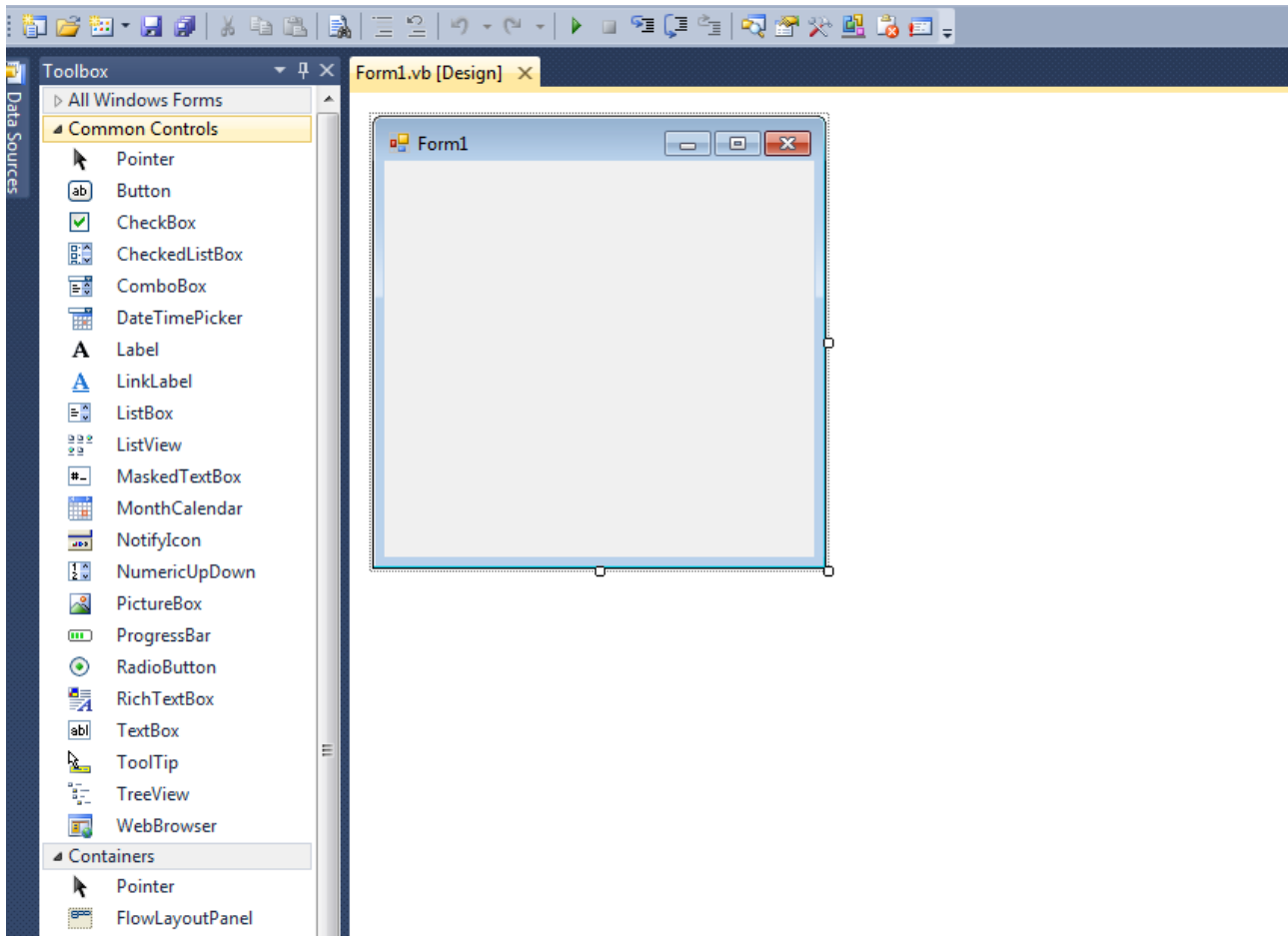
In SDD we will always be creating a “Windows Form Application”.



Make sure that you always rename the application, just like when you named your Xcode projects.

The Visual Basic IDE

Every screen in your Visual Basic program is known as a form – the equivalent of a scene in your storyboard. Unfortunately there is no storyboard facility in Visual Basic, so you manually need to manage switching between forms. We're not going to worry about that for the moment – we're going to focus on the basics of creating a single form application.

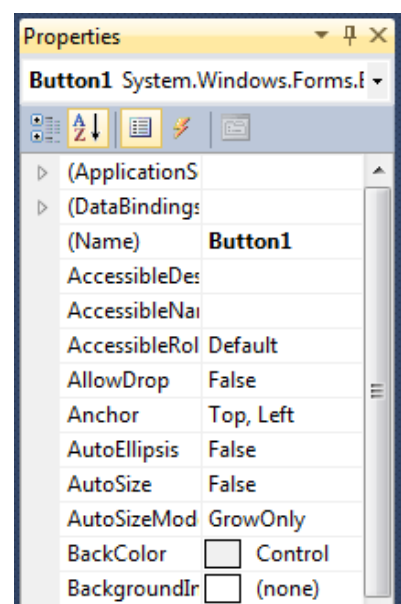


The left hand side of the screen contains the toolbox, with all the various user interface elements you can use. You can drag these onto the form, just like laying out a view on a storyboard. As an example, drag a Button onto the form.

Select the button and look at the right-hand side of the screen. You will see the Properties window for the selected control (in this case, the button). The first thing that you should do is rename the control to something more meaningful.

There are standard prefixes you should use for different types of controls (see the last page of this booklet). In the case of the button, it is *btn*. So the button should be named something like *btnDoStuff*.

You can go through the properties and modify other aspects, such as the Text that appears on the button itself (i.e. the label of the button).



Editing code

In Xcode and Swift we created IBActions to link user interface elements to code – this is a core feature of that language which separates content from presentation. This is not the case in Visual Basic, where the code and presentation are fairly well stuck together.

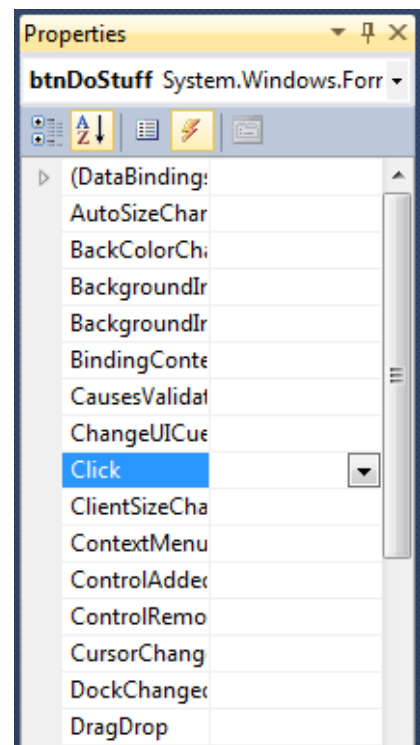
If you want to perform a particular action (i.e. run some code) in response to an event on a control, such as a button click, then select the control (e.g. button) and in the bottom right-hand corner of the screen select the lightning bolt in the properties window.

This will show all the events which can be fired for this particular control (again, similar to Xcode showing all the events for a particular control). In the case of a button, we are probably most interested in the *Click* event. Double-click in the drop-down box next to the *Click* event and this will open a code window.

Visual Basic automatically creates a method and links that to the *Click* event on the particular control. In other words, this is like Xcode automatically creating an IBAction method and then linking it to the relevant action on the control.

Any code that you want to run in response to that event can be placed in this method.

Pro Tip: If you double-click a control it will do the same thing, but automatically create the method and linked action for the most common action on that control. For example, if you double-click on a button it will create a method to respond to the button click event (since that's the most common method used for buttons). However, if you want to use any other methods you'll need to go to the lightning bolt instead.



Event handler methods

Methods which handle events (e.g. button click) are known as “event handler methods”. Look at the following code as an example:

```
Private Sub btnDoStuff_Click(sender As System.Object, e As System.EventArgs)
    Handles btnDoStuff.Click

    ' Put your code here

End Sub
```

The name of this method is `btnDoStuff_Click` and the method accepts two parameters/parameters:

- `sender` which is a reference to the object which called the method, just like in Swift.
- `e` which provides information about the event which occurred.

For the moment we’re not too concerned about either of those parameters – they are useful later, but not now.

We are far more interested in the `Handles btnDoStuff.Click` on the end of the method signature. This indicates the event(s) to which this method will run in response. In this case, it will run in response to the `Click` event on `btnDoStuff`. This is the equivalent of dragging a line to link the event in the storyboard to an `IBAction` in the code file.

Most of the time the default handler which Visual Basic provides is fine. However, you may occasionally want to link multiple handlers to a single method (i.e. have one method respond to events from different controls) in which case manually editing and adding handlers can be useful.

Declaring variables

Here are declarations for some common data/object types in **Swift**:

```
var myInteger: Int?  
var myFloat: Float?  
var myString: String?  
var myArrayOfStrings: Array<Int?>
```

Here are declarations for the same data types in **Visual Basic.NET**:

```
Dim myInteger As Integer  
Dim myFloat As Single  
Dim myString As String  
Dim myArray(5) As Integer
```

Differences to observe:

- Visual Basic does not have the concept of optionals, so forget about the ? and ! which you used in Swift.
- In contrast to Swift, all variables are variables. Visual Basic does not generally use constants (i.e. the equivalent of `let` in Swift).
- It is obviously a lot more verbose in Visual Basic, with all the `Dim` keyword stuff.
- Arrays in Visual Basic are fixed-length, so you need to define the length of the array in advance. More specifically, you need to specify the last index of the array. In the above example `Dim myArray(5) As Integer` you have created an array of length 6 with indexes 0...5 inclusive.

If you need to change the length of the array once it has been created, you can use the `ReDim` statement. For example, using `ReDim Preserve myArray(7)` you can change the length of the array to 8 items (indexes 0...7 inclusive). Be sure to include the `Preserve` keyword, which will preserve the data in the array.

Using variables

Here is an example of using the aforementioned variables in **Swift**:

```
myInteger = 5
myFloat = 5.8
myString = "Hello World"
myArray = ["String 1", "String 2", "String 3"];
```

Here is the equivalent in **Visual Basic.NET**:

```
myInteger = 5
myFloat = 5.8
myString = "Hello World"
myArray(0) = "Object 1"
myArray(1) = "Object 2"
myArray(2) = "Object 3"
```

Differences to observe:

- The myInteger, myFloat, and myString variables are practically the same for both languages.
- The main differences arise when we use complex data types, such as arrays – you need to access each item via its index.

Selection (if) statements – binary selection

Binary (if-else) selection in **Swift**:

```
var myInteger = 5

if myInteger < 5 {
    print("My integer is less than 5")
}

else {
    print("My integer is greater than 5")
}
```

The equivalent in **Visual Basic**:

```
Dim myInteger As Integer = 5

If (myInteger < 5) Then
    ' Show a message box - halts execution of the program until dismissed
    MsgBox("My integer is less than 5")
    ' Print to the immediate window - does not halt execution (similar to print in Swift)
    Debug.Print("My integer is less than 5")
Else
    MsgBox("My integer is greater than 5")
    Debug.Print("My integer is greater than 5")
End If
```

The above has introduced a few new concepts, in addition to the if statement construct:

- Single-line comments in Visual Basic are preceded with a single quote ' mark. There is no equivalent for block comments.
- To output debug messages there are two common ways: either use MsgBox or Debug.Print.
 - MsgBox will produce a message box on screen and will halt execution of the program until the user dismisses (i.e. clicks OK – a bit like a UIAlertView).
 - Debug.Print will write the output to the immediate window in Visual Basic – this is similar to print() in Swift.

Selection (if) statement – multiway selection

Multiway (if-elseif-else) selection in **Swift**:

```
var myInteger = 5

if myInteger < 5 {
    print("My integer is less than 5")
}

else if myInteger < 10 {
    print("My integer is less than 10")
}

else {
    print("My integer is greater than 10")
}
```

The equivalent in **Visual Basic**:

```
Dim myInteger As Integer = 5

If (myInteger < 5) Then
    Debug.Print("My integer is less than 5")
ElseIf (myInteger < 10) Then
    Debug.Print("My integer is less than 10")
Else
    Debug.Print("My integer is greater than 10")
End If
```


Repetition statements – counted loops

Counted loop in **Swift**:

```
var myInteger = 5

for i in 0...myInteger {
    print(i)
}
```

The equivalent in **Visual Basic**:

```
Dim myInteger As Integer = 5

For i As Integer = 0 To myInteger Step 1
    Debug.Write(i)
Next
```

Step indicates by how much the loop counter should be incremented each time the loop is run. If the step is 1 it can be omitted and it will be assumed to be 1 – it is included above for demonstration purposes only, so the following would be equivalent:

```
Dim myInteger As Integer = 5

For i As Integer = 0 To myInteger
    Debug.Write(i)
Next
```

You should always use a counted loop when possible, particularly when writing pseudocode in exams. Many students have lost easy marks in the past due to using a while loop and then forgetting to do something like increment the loop counter – if you use a counted loop then you won't make this mistake, as everything is built into the loop construct.

Repetition statements – pre-test loops

Pre-test loop in **Swift**:

```
var i = 0

while i <= 5 {
    print(i)
    i = i + 1
}
```

The equivalent in **Visual Basic**:

```
Dim i As Integer = 0

While (i <= 5)
    Debug.Write(i)
    i = i + 1
End While
```

Repetition statements – post-test loops

Post-test loop in **Swift**:

```
int i = 0;

repeat {
    print(i)
    i = i + 1;
} while i <= 5
```

The equivalent in **Visual Basic**:

```
Dim i As Integer = 0

Do
    Debug.Write(i)
    i = i + 1
Loop While (i <= 5)
```

Custom Methods (Subs) – no parameters

Custom method in **Swift**

```
func myCustomMethod() {  
  
    // Do stuff here  
  
}
```

Calling the method (from the same class) in **Swift**:

```
myCustomMethod()
```

The equivalent in **Visual Basic**:

```
Private Sub myCustomMethod()  
  
    ' Do stuff here  
  
End Sub
```

Calling the method (from the same class) in **Visual Basic**:

```
myCustomMethod()
```

Custom Methods (Subs) – with parameters

Custom method in **Swift**:

```
func myCustomMethod(paramOne, paramTwo) {  
  
    // Do stuff here  
  
}
```

Calling the method (from the same class) in **Swift**:

```
myCustomMethod("Hello", "World")
```

The equivalent in **Visual Basic**:

```
Private Sub myCustomMethod(paramOne As String, paramTwo As String)  
  
    ' Do stuff here  
  
End Sub
```

Calling the method (from the same class) in **Visual Basic**:

```
myCustomMethod("Hello", "World")
```

Custom Functions

Custom function in **Swift**:

```
func myCustomFunction(paramOne) -> Int {  
  
    return 1;  
  
}
```

Calling the method (from the same class) in **Swift**:

```
var returnValue = myCustomFunction("Hello")
```

The equivalent in **Visual Basic**:

```
Private Function myCustomFunction(paramOne As String) As Integer  
  
    Return 1  
  
End Function
```

Calling the method (from the same class) in **Visual Basic**:

```
Dim returnValue As Integer = myCustomFunction("Hello")
```

Classes and Objects in Visual Basic

Okay, so there may be situations where you do want to use classes and objects in Visual Basic. For example, you may want to use them in your major projects (otherwise you could have very messy code). Alternatively, you may want to use some in-built classes – the most common being an `ArrayList` (which is essentially a variable-length array, if you don't want to mess around with `ReDim` statements).

NEVER use classes/objects in HSC SDD exams. They are not part of the syllabus and are only useful for your own internal school programming projects.

Here's an example of creating an object from a `CustomArray` class in **Swift**:

```
var customArray = CustomArray()
```

Here is the equivalent in **Visual Basic** (using the `ArrayList` class):

```
Dim myArrayList As New ArrayList
```

The creation of an object from a class in Visual Basic is simply done by inserting the `New` keyword before the data type in the variable declaration (`Dim`) statement.

Calling a method – with no parameters

To call a method on an object you would do the following in **Swift**:

```
customArray.count()
```

Here is the equivalent way to call a method on an object in **Visual Basic**:

```
myArrayList.Count()
```

If the method takes no parameters/inputs then the brackets are optional, so the following is equivalent: `myArrayList.Count`

Calling a method – with one parameter

To call a method which takes one parameter in **Swift** you would do the following:

```
customArray.append("Item 1")
```

Here is the equivalent way in **Visual Basic**:

```
myArrayList.Add("Object 1")
```

Calling a method – with multiple parameters

To call a method which takes multiple parameters in **Swift**:

```
customArray.insert("Object", atIndex:5)
```

In this example you can see a nice feature of Swift, it's a very self-documenting language. It is clear what each parameter into the method does: the first parameter is the object to be inserted, and the second parameter is the index in which to insert the object.

Here is the equivalent way in **Visual Basic**:

```
myArrayList.Insert(5, "Object")
```

Unfortunately it is less clear in Visual Basic as to the purpose of each parameter. What does the 5 mean? Is that the index? You would need to look at the Visual Basic documentation to be certain. It's not just Visual Basic that has this limited, but other languages such as PHP and Java also have this issue.

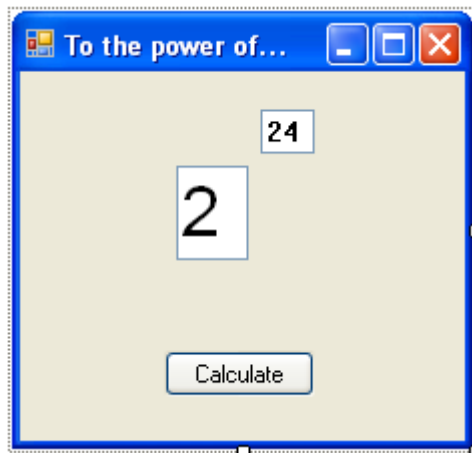
Activities

1. To the power of...

You need to write a program that allows the user to enter a base number and exponent. When the user clicks the calculate button it will display (in a message box) the result of the calculation. For example, in the above program it should calculate $2^{24} = 16777216$. Therefore, the number “16777216” should be displayed in the message box.

Steps that you will need to undertake to make the solution:

- (a) Design the form with all the required controls. You will need two text boxes and a button.
- (b) Modify the properties of each control so they appear as above. You will need to change the text on the button, change the font on the text boxes (so that the font size is larger), and rename the button, text boxes, and the form following the appropriate conventions (for example, *frmMainForm* for the form, *btnCalculate* for the button, etc).
- (c) Once you have designed the form, you need to write the code which runs when clicking on the Calculate button. You will need to write the code within the `_Click` event for the Calculate button. You can open the code for this event by either double-clicking the button (remember that double-clicking on a control will open the code for the most common event which that control performs) or you can click the events (lightning-bolt) icon in the Properties sub-window in Visual Basic and then double-click on the “Click” event.
- (d) It is always easier to write what you want the code to do in English (or pseudocode) before implementing the solution in programming code, as this ensures that you plan out the logic of your code. For example, the code for this solution must follow this logic:
 - a. Declares a variable to store the calculated number (perhaps an Integer?)
 - b. Performs the required calculation and stores this result in the variable declared in part (a). Hint – the operator for calculating the “power of” in Visual Basic is the `^` character. Using the above example, to calculate 2^{24} you would write `2^24`.
 - c. Displays the number stored in the variable within a message box.



Extension Activities:

- (a) If your program does not already, modify the program so that you can complete the calculation with non-integer numbers (i.e. numbers with a decimal point).
- (b) **(Fun Size) Mars Bar Challenge.** Modify the program so that you can only enter numbers and valid characters: 0 – 9, negative sign (-), and a decimal point (.). Any other input will be disregarded and not entered into the field.

2. Odd or Even?

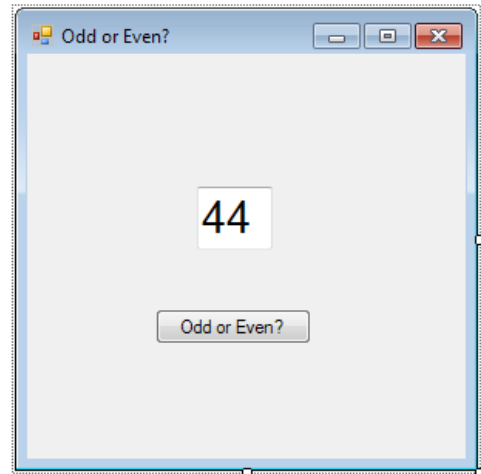
You need to write a program which will allow the user to enter a number and then determines whether that number is an odd or even number. Once the user enters a number and clicks the “Odd or Even?” button a message box will appear informing whether that number is odd or even.

Some helpful hints for this exercise:

- (i) The interface design consists of one text box and one button, which runs the relevant code (to check whether the number is odd or even) when clicked.
- (ii) You will need to use selection (`If`) statements within the program to perform a different action (i.e. a message box informing whether the number is odd or even) depending on whether the number is odd or even. The format of the `If` statement is as follows:

```
If <number is odd> Then  
    show message box informing that the number is odd  
Else  
    Show message box informing that the number is even  
End If
```

- (iii) To access the text which has been inputted into a text field, use the `.text` property of the text field.
- (iv) The `mod` mathematical operator can be used to determine whether a number is odd or even. This is also known as the modulo operation (http://en.wikipedia.org/wiki/Modulo_operation) which determines the remainder from a division operation. `mod` will return 1 if there is a remainder, and 0 if there is no remainder. Therefore, if you use `mod` with the number 2 a returned value of 1 will indicate that the number does not divide exactly by 2 and is therefore an odd number. Conversely, a returned value of 0 will indicate that the number does divide exactly by two (as there is no remainder) and thus it is even. For example, $5 \bmod 2 = 1$ whereas $8 \bmod 2 = 0$



Extension Activities:

- (a) Modify the program so that after the original message box appears, informing whether the entered number is odd or even, another message box appears which contains the sum of all the numbers up to and including the entered number. **Hint:** For this activity you will need to write a loop.
- (b) Modify the program so that after the two existing message boxes appear, another message box appears with the nearest prime number to the entered number.

3. Fizz Buzz Test

Write a program that prints the numbers from 1 to 100. But for multiples of three print “Fizz” instead of the number and for the multiples of five print “Buzz”. For numbers which are multiples of both three and five print “FizzBuzz”.

Source: <http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/>

All the output should be printed in the Immediate Window in Visual Basic.

You should only need to have a single button on your form, within which you can write all your code to run the Fizz Buzz test.

4. Times Tables

You need to write an application where the user can type a number and it will calculate the times tables (up to $\times 12$) for that number. In this version of the program each number in the times table should be printed to the Immediate Window in the following format:

$$\text{Base Number} \times \text{Multiplier} = \text{Result}$$

For example, if the program is calculating the 2 times table, and it is currently on the third iteration, then the following will be displayed in the Immediate Window:

$$2 \times 3 = 6$$

Then, once the user clicks OK, to dismiss that message box, on the next iteration the following will be displayed in the window:

$$2 \times 4 = 8$$

... and so on, until the program reaches $2 \times 12 = 24$, after which the program terminates.

You should follow the same design methodology as for Exercise 1, designing the interface and modifying the properties of all the elements before thinking about the code. Then, when you are ready to start work on the code write out the algorithm you will be coding in plain English, so you know what you want to do, before implementing it in programming code.

Some helpful hints for this exercise:

- (i) The interface design portion is pretty much the same as Exercise 1, except you only have one text box instead of two! If you successfully completed the interface design in Exercise 1 then you shouldn't have any problems with this part.



Continued on next page →

- (ii) You will need to use a loop within this program. More specifically, you will need to write a loop which loops 12 times from 1 to 12 (inclusive), each time multiplying the loop iteration counter against the number for which you are calculating the times table. The format for the loop is as follows:

```
While <condition is true>
    do cool stuff here
End while
```

More specifically, you will first need to declare a variable to count the number of times you have run the loop.

```
Dim i As Integer
```

Then, you need to write the loop itself. In this case, we want to iterate 12 times starting from 1 and then finishing after the number 12 has been processed.

```
` Notice that we have initialized i as 1 since we want to loop
` starting on the number 1
```

```
i = 1
```

```
While i <= 12
```

```
` do stuff here
```

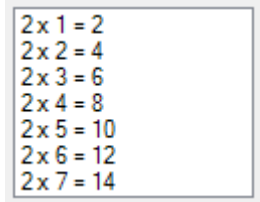
```
End while
```

The code above will loop from 1 to 12 inclusive. We have specifically declared the loop condition as while `i` is **less than or equal to** 12 so that it will run on the 12th iteration and not stop one short.

Now you can complete the bit within the loop. Remember, you just need to take the number entered into the text box on the form and then multiply by `i` on each loop iteration, and display the result in a message box. This is a bit like what you did in Exercise 1 when calculating the power of a particular number.

Extension Activities:

- (a) Add a list box to the form. Instead of displaying each of the outputs from the program in an individual message box, add them to the list box as a new row. Hint: To add elements to a list box, use: **ListBoxName.Items.Add(StringToAddHere)**



2x1=2
2x2=4
2x3=6
2x4=8
2x5=10
2x6=12
2x7=14

- (b) Write the results of the list box to a text file. The contents of the text file should look exactly the same as the list box, with each result printed on a separate line. Hint: To write elements out to a text file look at the “Write a text file” section of the following Microsoft Knowledge Base article: <http://support.microsoft.com/kb/304427> . You will just need to call the `.WriteLine` function multiple times, to write each line of the list box to the text file. You can retrieve a particular row of a list box using **ListboxName.Items.Item(IndexOfItem)**

VB.net naming conventions

Control Type	prefix
Button	btn
Checkbox	chk
CheckedListBox	clst
ComboBox	cbo
ContextMenu	cmnu
DataGrid	grd
DateTimePicker	dtp
DomainUpDown	dup
Form	frm
GroupBox	grp
HelpProvider	hlp
HScrollBar	hscr
ImageList	ilst
Label	lbl
LinkLabel	lnk
ListBox	lst
ListView	lvw
MainMenu	mnu
MonthCalendar	cal
NotifyIcon	nic
NumericUpDown	nup
OpenFileDialog	ofd
Panel	pnl
PictureBox	pic
ProgressBar	prg
RadioButton	rdo
RichTextBox	rtxt
SaveFileDialog	sfd
Splitter	spl
StatusBar	sbar
TabControl	tab
TextBox	txt
Timer	tim
ToolBar	tbar
ToolTip	tip
TrackBar	trk
TreeView	tvw
VScrollBar	vscr

Reference: <http://www.pdsa.com/Tips/DotNetProgrammingStdts.doc>