Microsoft Office Training Series

Access

VBA Introduction

- Courses never Cancelled
- 12+ Months Schedule
- 24 Months Online Support
- UK Wide Delivery

STL MicrosoftTraining.net

Learning & Performance Institute
Accredited Learning Provider

ilm Recognised Provider

Microsoft Partner
Certified Silver Partner

WELCOME TO YOUR ACCESS VBA INTRODUCTION TRAINING COURSE

- The VBA environment
- Create Sub procedures and Functions
- Understand Object in Object Oriented Programming
- Understand Variables and Constants
- Loops & Decision code
- Error Handling and Debugging
- Working with recordset (ADO & DAO)

CONTENTS

Unit 1 The VBA Environment

Introducing Visual Basic For Applications

Visual Basic for Applications or VBA is a development environment built into the Microsoft Office© Suite of products.

VBA is an Object Oriented Programming (OOP) language. It works by manipulating objects. In Microsoft™ Office® the programs are objects.

In Access worksheets, tables, forms, reports and queries are also objects.

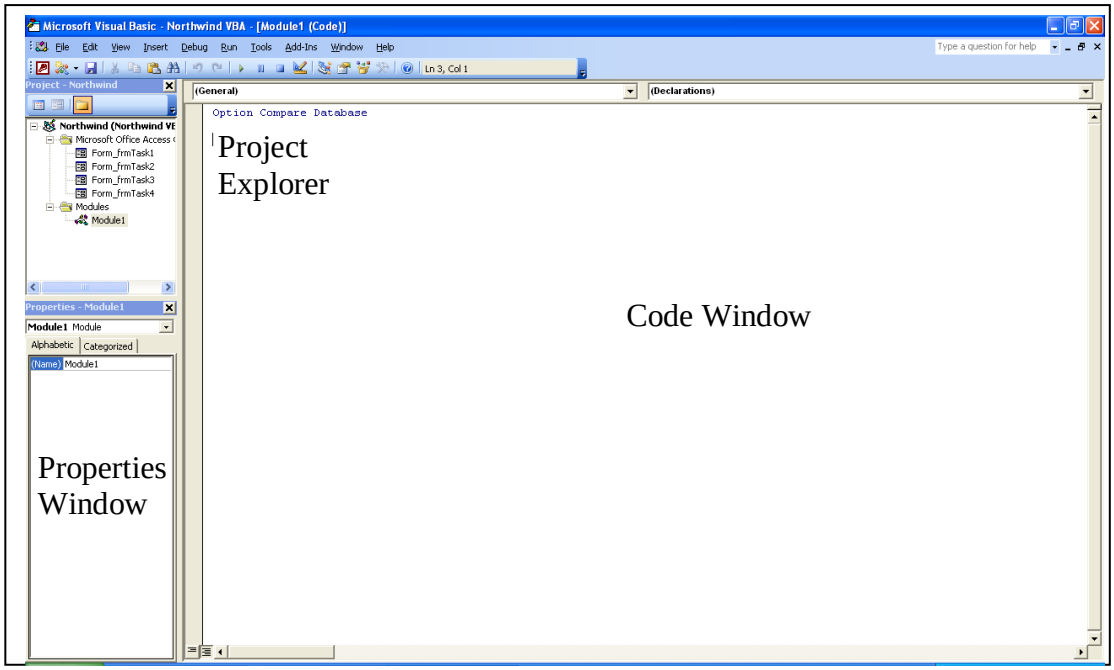
In VBA the object is written first

I'm fixing house number 42 = .House.42.Fix

	House	**42**	**Fix**
English	.noun	.noun	.verb
VBA	.object	.child object	.method

When working in VBA tell Access exactly what to do. Don't assume anything.

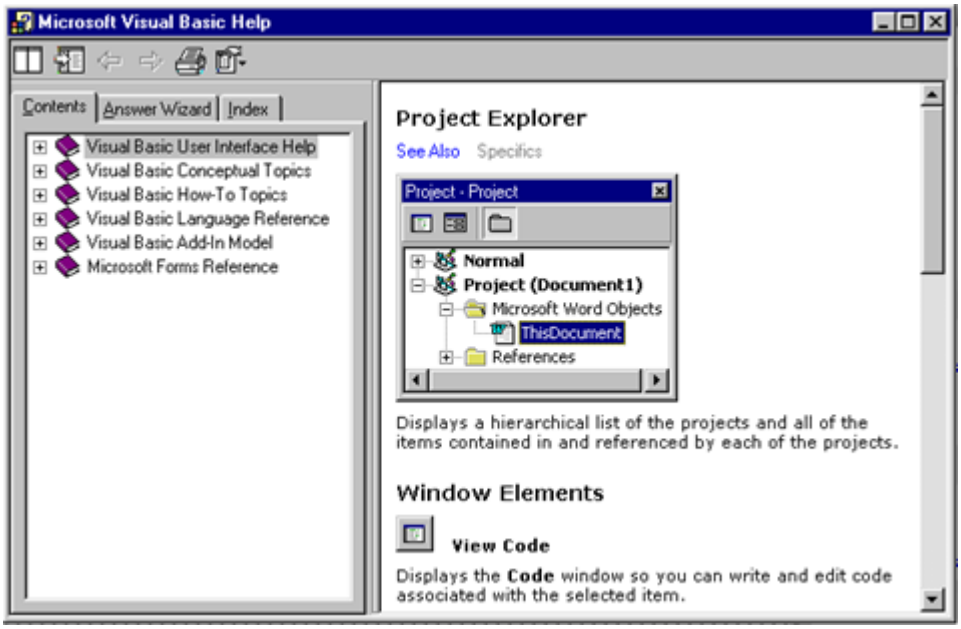
Understanding the Development Environment



Title bar, Menu bar and Standard toolbar	The centre of the Visual basic environment. The menu bar and toolbar can be hidden of customized. Closing this window closes the program.
Project Explorer	Provides an organized view of the files and components belonging to the project. If hidden the Project Explorer can be displayed by pressing **Ctrl + R**
Properties Window	Provides a way to change attributes of forms and controls (e.g. name, colour, etc). If hidden press **F4** to display.
Code Window	Used to edit the Visual basic code. Press **F7** and it will open an object selected in Project Explorer. Close the window with the **Close** button that appears on the menu bar.

Using Help

If the **Visual Basic Help** files are installed, by pressing **F1**, a help screen displays explaining the feature that is currently active:



Alternatively use the **Ask a Question** box on the menu bar to as a quick way to find help on a topic.

Closing the Visual Basic Editor

To close the **Visual Basic Editor** use one of the following:

- Open the **File** menu; select **Close and Return to Microsoft Access**

OR

- Press **Alt + Q**

OR

- Click ✕ **Close** in the title bar.

Unit 2 Developing with Procedures and Functions

Procedure is a term that refers to a unit of code created to perform a specific task. In Access, procedures are stored in objects called **Modules**.

In this unit we will look at both Modules and Procedures.

Understanding and Creating Modules

Standard modules can be used to store procedures that are available to all objects in your application

Within a project you can create as many standard modules as required. You should store related procedures together within the same module.

Standard modules are also used to declare global variables and constants.

To create a standard module in the VB Editor:

- Display the **Properties** window if necessary
- In the **Properties** window change the name of the module

Defining Procedures

A procedure is a named set of instructions that does something within the application.

To execute the code in a procedure you refer to it by name from within another procedure. This is known as Calling a procedure. When a procedure has finished executing it returns control to the procedure from which it was called.

There are two genera types of procedures:

Sub procedures	perform a task and return control to the calling procedure
Function procedures	perform a task and return a value, as well as control, to the calling procedure

If you require 10 stages to solve a problem write 10 sub procedures. It is easier to find errors in smaller procedures than in a large one.

The procedures can then be called, in order, from another procedure.

Naming Procedures

There are rules and conventions that must be followed when naming procedures in Visual Basic.

While rules must be followed or an error will result, conventions are there as a guideline to make your code easier to follow and understand.

The following **rules** must be adhered to when naming procedures:

- Maximum length of the name is 255 characters
- The first character must be a letter
- Must be unique within a given module
- Cannot contain spaces or any of the following characters: **. , @ & $ # ( ) !**

You should consider these naming **conventions** when naming procedures:

- As procedures carry out actions, begin names with a verb
- Use the proper case for the word within the procedure name
- If procedures are related try and place the words that vary at the end of the name

Following these conventions, here is an example of procedure names:

PrintClientList

GetDateStart

GetDateFinish

Creating a Sub-Procedure

Most Access tasks can be automated by creating procedures. This can be done by either recording a macro or entering the code directly into the VB Editor's Code window.

Sub procedures have the following syntax:

```
[Public/Private] Sub ProcedureName ([argument list])
Statement block
End Sub
```

Public indicates procedure can be called from within other modules. It is the default setting

Private indicates the procedure is only available to other procedures in the same module.

The **Sub…End Sub** structure can be typed directly into the code window or inserted using the **Add Procedure** dialog box.

To create a sub procedure:

- Create or display the module to contain the new sub procedure
- Click in the **Code** window

- Type in the Sub procedure using the relevant syntax

Type in the word Sub, followed by a space and the Procedure name
Press **Enter** and VB inserts the parenthesis after the name and the End Sub line.

Below is an example of a basic sub procedure:

```
Sub Welcome()
    MsgBox "Hello User, How are you"
End Sub
```

Creating a Function Procedure

Function procedures are similar to built-in functions such as DateDiff(). They are sometimes called **user-defined** function.

A function returns a value to the procedure that calls it. The value the function generates is assigned to the name of the function.

Function procedures have the following syntax:

[Public/Private] Function FunctionName ([argument list]) [As <Type>]

[Statement block]

[FunctionName = <expression>]

End Function

Public indicates procedure can be called from within other modules. It is the default setting

Private indicates the procedure is only available to other procedures in the same module.

The **As** clause sets the data type of the function's arguments and return value.

To create a function procedure:

- Create or display the module to contain the new Function procedure
- Click in the **Code** window
- Type in the Function procedure using the relevant syntax or use **Add Procedure**

Type in the word Function followed by a space and the Function name
Press **Enter** and VB places the parenthesis after the name and inserts the End Function line.

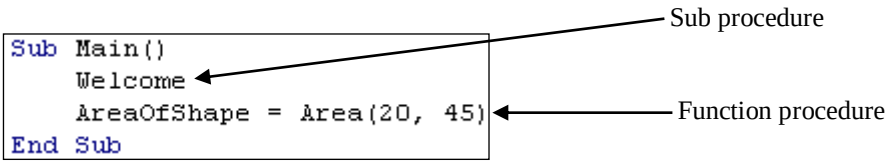
Below is an example of a basic function procedure:

```
Function Area(Length As Integer, Width As Integer) As Integer
    Area = Length * Width
End Function
```

Calling Procedures

A sub procedure or function is called from the point in another procedure where you want the code to execute. The procedure being called must be accessible to the calling procedure. This means it must be in the same module or be declared public.

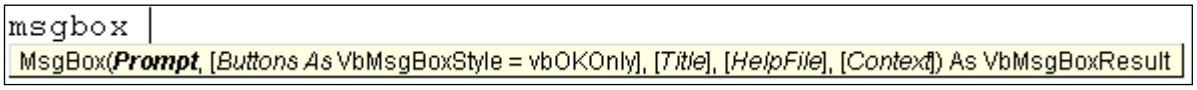
Below is an example of calls to Sub and Function procedures:



When passing multiple arguments (as in the function procedure above) always separate them with commas and pass them in the same order as they are listed in the syntax.

Auto Quick Info is a feature of the Visual Basic that displays a syntax box when you type a procedure or function name.

The example below shows the tip for the Message Box function:



Arguments in square brackets are optional.

Values passed to procedures are sometimes referred to as parameters.

Function Examples

Create an AGE function:
If you need to calculate age in a query or in a form an AGE function can be useful. The Int function is used in the example to return a whole number.

Type in a module:

```
Function Age(StartDate)

Age=Int((Date-StartDate)/365.25)

End Function
```

Function with Optional Arguments

If we have a hire date for staff members and some of them also have an end date (They have left the company). We can amend the Age function with an Optional Argument. An If decision code is used to tell the function how to use the Optional argument.

```
Function Age(StartDate, Optional EndDate)

If IsEmpty(EndDate) = True Then
        Age = Int((Date - StartDate) / 365.25)
Else
        Age = Int((EndDate - StartDate) / 365.25)
End If

End Function
```

Show an address on Google map

The function below can be called from a Sub procedure from which it receive the address, postcode, city and country.

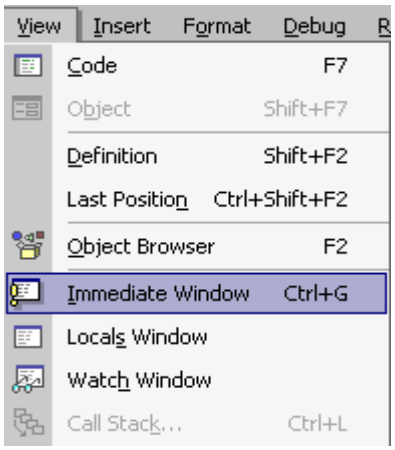
```
Function MapAddress(Address, PostCode,City, Country)

Dim Address as String

Address=Address&" "&PostCode&" "&City&" "&Country

Application.FollowHyperlink "http://www.google.co.uk/maps/search/"&Address

End Function
```

Using the Immediate Window to Call Procedures

The **Immediate window** is a debugging feature of Visual Basic. It can be used to enter commands and evaluate expressions.

Code stored in a sub or function procedure can be executed by calling the procedure from the **Immediate window**.

To open the **Immediate window**:



- Open the **View** menu
- Select **Immediate window**

OR

- Press **Ctrl+G**.

The **Immediate window** appears.

To execute a sub procedure:

- Type **SubProcedureName ([Argument list])**
- Press **Enter**.

To execute a function and print the return value in the window:

- Type **? FunctionName ([Argument list])**
- Press **Enter**.

To evaluate an expression:

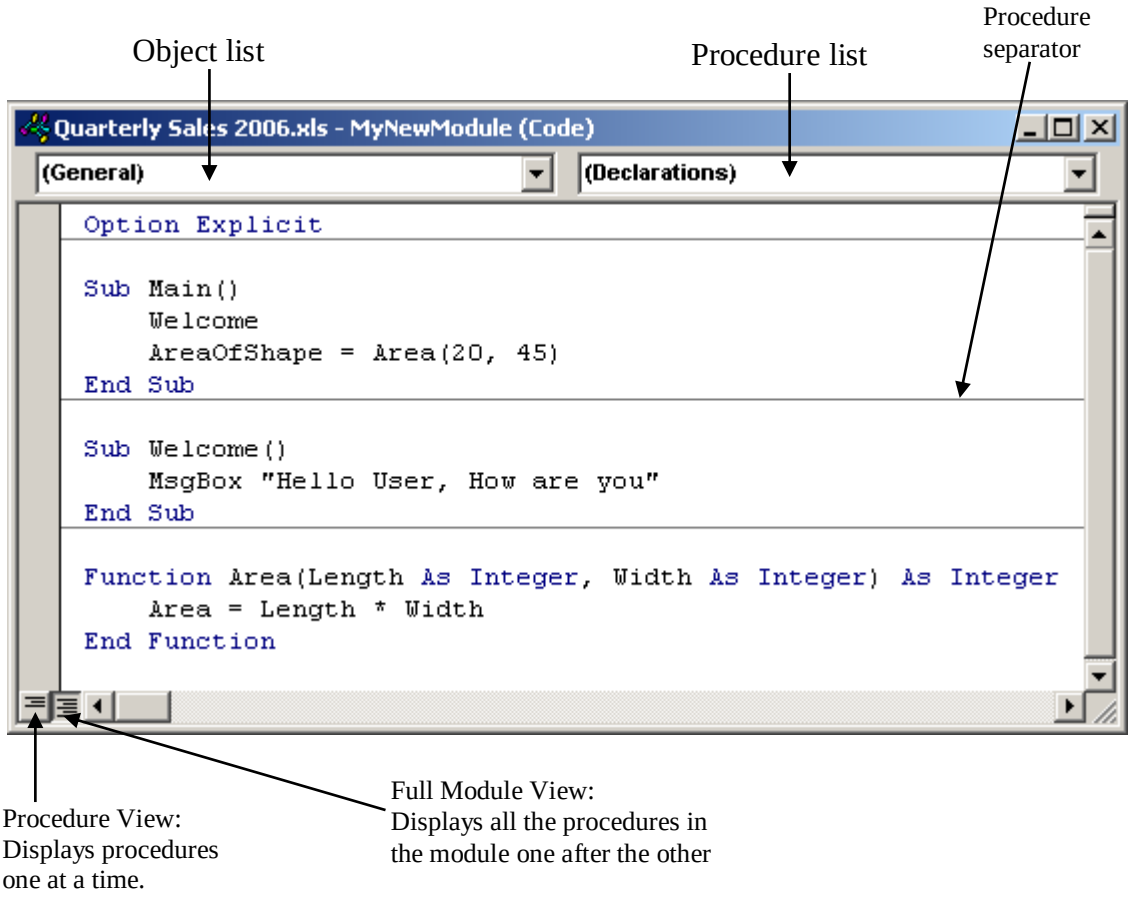
- Type **? Expression**
- Press **Enter**.

Within the code, especially in loops, use the **Debug.Print** statement to display values in the Immediate window while the code is executing. The Immediate window must be open for this.

Working Using the Code Editor

The Code Editor window is used to edit Visual Basic code. The two drop down lists can be used to display different procedures within a standard module or objects' event procedures within a class module.

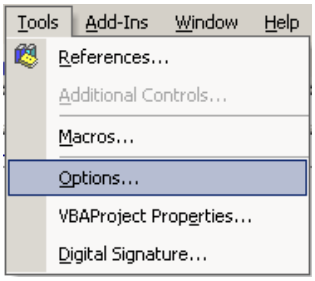
Below is an illustration of the code window:



Object List	Displays a list of objects contained in the current module.
Procedure List	Displays a list of general procedures in the current module when General is selected in the Object list. When an object is selected in the Object list it displays a list of events associated with the object.

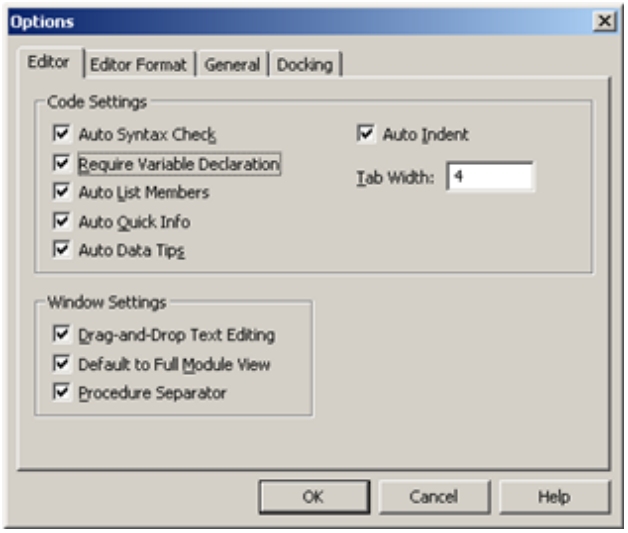
Setting Code Editor Options

The settings for the **Code Editor** can be changed. To do this:



- Open the **Tools** menu in the **VB Editor**
- Select **Options**.

The **Options** dialog box appears:



The following are explanations of the **Code Setting** selections:

Auto Syntax Check	Automatically displays a **Help** message when a syntax error is detected. Message appears when you move off the code line containing the error
Require Variable Declaration	Adds the line ***Option Explicit*** to all newly created modules, requiring all variables to be explicitly declared before they are used in a statement.
Auto List Members	Displays a list box under your insertion point after you type an identifiable object. The list shows all members of the object class. An item selected from the list can be inserted into your code by pressing the **Tab** key
Auto Quick Info	Displays a syntax box showing a list of arguments when a method, procedure or function name is typed
Auto Data Tips	Displays the value of a variable when you point to it with a mouse

during break mode. Useful for debugging.

Auto Indent Indent the specified amount when **Tab** is pressed and indents all subsequent lines at the same level.

The **Windows Settings** selections are explained below:

Drag-and-Drop Text Editing	Allows you to drag and drop code around the Code window and into other windows like the Immediate window.
Default to Full Module View	Displays all module procedures in one list with optional separator lines between each procedure. The alternative is to show one procedure at a time, as selected through the Procedure list.
Procedure Separator	Displays a grey separator line between procedures if Module view is selected

Editing Guidelines

Below are some useful guidelines to follow when editing code:

- If a statement is too long carry it over to the next line by typing a space and underscore (_) character at the end of the line. This also works for comments.

Strings that are continued require a closing quote, an ampersand (&), and a space before the underscore. This is called ***Command Line Continuation***.

- Indent text within control structures for readability. To do this:

- Select one or more lines
- Press the **Tab** key **OR**
- Press **Shift + Tab** to remove the indent.

- Complete statements by pressing **Enter** or by moving focus off the code line by clicking somewhere else with the mouse or pressing an arrow key.

When focus is moved off the code line, the code formatter automatically places key words in the proper case, adjusts spacing, adds punctuation and standardizes variable capitalization.

It is also a good idea to comment your code to document what is happening in your project. Good practice is to comment what is not obvious.

Start the line with an apostrophe (') or by typing the key word **Rem** (for remark). When using an apostrophe to create a comment, you can place the comment at the end of a line containing a code statement without causing a syntax error.

Unit 3 Understanding Objects

An object is an element of an application that can be accessed and manipulated using Visual Basic.

Defining Objects

Objects are defined by lists of **Properties**, and **Methods**. Many also allow for custom sub-procedures to be executed in response to **Events**.

The term **Class** refers to the general structure of an object. The class is a template that defines the elements that all objects within that class share.

Properties

Properties are the characteristics of an object. The data values assigned to properties describe a specific instance of an object.

A new Form in Access is an instance of a Form object, created by you, based on the Form class. Properties that define an instance of a Form object would include its Name, Caption, Size, etc.

Methods

Methods represent procedures that perform actions.

Printing a report, updating a record and running a query are all examples of actions that can be executed using a method.

Events

Many objects can recognize and respond to events. For each event the object recognizes you can write a sub procedure that will execute when the specific event occurs.

A Form recognizes the Open event. Code inserted into the Open event procedure of the Form will run whenever the Form is opened.

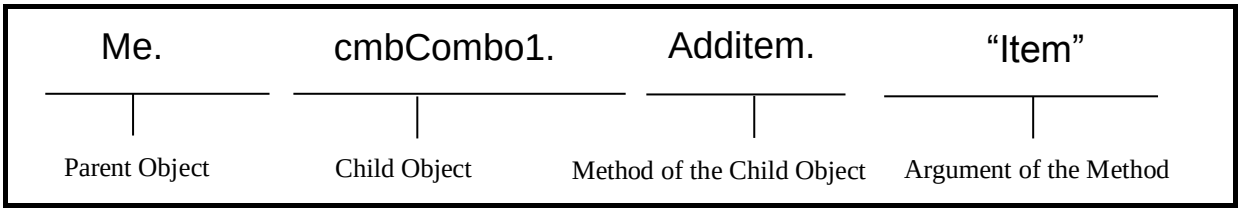
Events may be initiated by users, other objects, or code statements. Many objects are designed to respond to multiple events.

Examining the Access Object Hierarchy

The Access Object Module is a set of objects that Access exposes to the development environment. Many objects are contained within other objects. This indicates a hierarchy or parent-child relationship between the objects.

The Application object represents the application itself. All other objects are below it and accessible through it. It is by referencing these objects, in code, that we are able to control Access.

Objects, their properties and methods are referred to in code using the "dot" operator as illustrated below:



Some objects in the Access Object model represent a **Collection** of objects. A collection is a set of objects of the same type.

The Forms collection in Access represents a set of all open Forms. An item in the collection can be referenced using an index number or its name.

To view the entire Access Object model:

- Open the **Help** window
- Select the **Contents** tab
- Expand **Programming Information**
- Expand **Microsoft Access Visual basic Reference**
- Select **Microsoft Access Object Model**.

Defining Collections

A collection is a set of similar objects such as all open databases.

Many Access collections have the following properties:

Application Refers to the application that contains the collection

Count An integer value representing the number of items in the collection.

Item Refers to a specific member of the collection identified by name or position. Item is a method rather than a property

Some collections provide methods similar to the following:

Add Allows you to add items to a collection

Delete Allows you to remove an item from the collection by identifying it by name or position.

Referencing Objects in a Collection

A large part of programming is referencing the desired object, and then manipulating the object by changing its properties or using its methods. To reference an object you need to identify the collection in which it's contained.

The following syntax references an object in a collection by using its position. Since the **Item** property is the default property of a collection there is no need to include it in the syntax.

CollectionName(Object Index Number)

Forms(1)
Reports(7)

The following syntax refers to an object by using the object name. Again the **Item** property is not necessary:

CollectionName(ObjectName)

Forms("Employees")

Reports("Sales Report")

Using The Object Browser

The Object Browser is used to examine the hierarchy and contents of the various classes and modules.

The Object Browser is often the best tool to use when you are searching for information about an object such as:

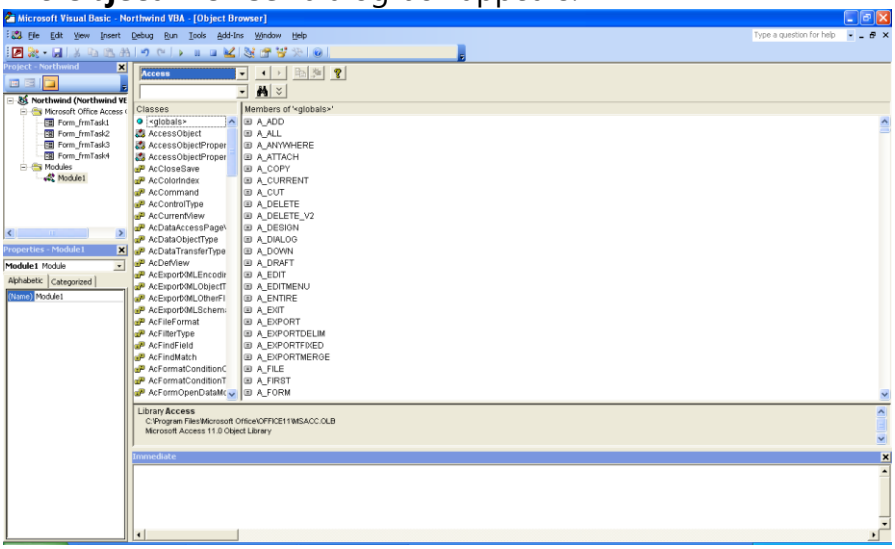
- Does an object have a certain property, method or event
- What arguments are required by a given method
- Where does an object fit in the hierarchy

To access the **Object Browser**:

In the **Visual Basic Editor**, do one of the following:

- Open the **View** menu
- Select **Object Browser** **OR**
- Press **F2** **OR**
- Click  the **Object Browser** icon.

The **Object Browser** dialog box appears.



The following icons and terms are used in the **Object Browser**:

	Class	Indicates a Class (Eg Form)
	Property	Is a value representing an attribute of a class (Eg. Name, Value)
	Method	Is a procedure that perform actions (Eg. Copy, Print Out, Delete)
	Event	Indicates an event which the class generates (Eg Click, Activate)
	Constant	Is a variable with a permanent value assigned to it (Eg vbYes)
	Enum	Is a set of constants
	Module	Is a standard module

To search for an object in the **Object Bowser**:

- Type in the search criteria in the Search Text box
- Click 

To close the Search pane:

- Click 

Working With Properties

Most objects in Access have an associated set of properties. During execution, code can read property values and in some cases, change them as well.

The syntax to read an object's property is as follows:

ObjectReference.PropertyName

Form.Name

The syntax to change an object's property is as follows:

ObjectReference.PropertyName = expression
Form.Name = "Quarterly Sales 2006"

The With Statement

The **With statement** can be used to work with several properties or methods belonging to a single object without having to type the object reference on each line.

The **With statement** helps optimize the code because too many "dots" in the code slows down execution.

The syntax for the **With statement** is as follows:

```
With ObjectName
    <Statement>
End With

With recordset
        .movefirst
        .movenext
        .edit
End With
```

You can nest **With statements** if needed.

Make sure that the code does not jump out of the **With** block before the **End With** statement executes. This can lead to unexpected results.

Working with Methods

Many Access objects provide public **Sub** and **Function** procedures that are callable from outside the object using references in your VB code. These procedures are called **methods**, a term that describes actions an object can perform.

Some methods require arguments that must be supplied when using the method.

The syntax to invoke an object method is as follows:

ObjectReference.Method [argument]

Forms(2).Open

Recordset.movelast

When calling procedures or methods that have arguments you have two choices of how to list the argument values to be sent.

Values can be passed by listing them in the same order as the argument list. This is known as a **Positional Argument**.

Alternatively you can pass values by naming each argument together with the value to pass. This is known as a **Named Argument**. When using this method it is not necessary to match the argument order or insert commas as placeholders in the list of optional arguments

The syntax for using named arguments is as follows:

Argumentname:= value

The example shows the **PrintOut** method and its syntax:

Sub PrintOut([From],[To],[Copies],[Preview],[ActivePrinter],[PrintToFile],[Collate], [PrToFilename])

The statements below show both ways of passing values when calling the PrintOut method. The first passes by **Position**, the second by **Naming**.

Event Procedures

An event procedure is a sub procedure created to run in response to an event associated with an object. For example run a procedure when a workbook opens.

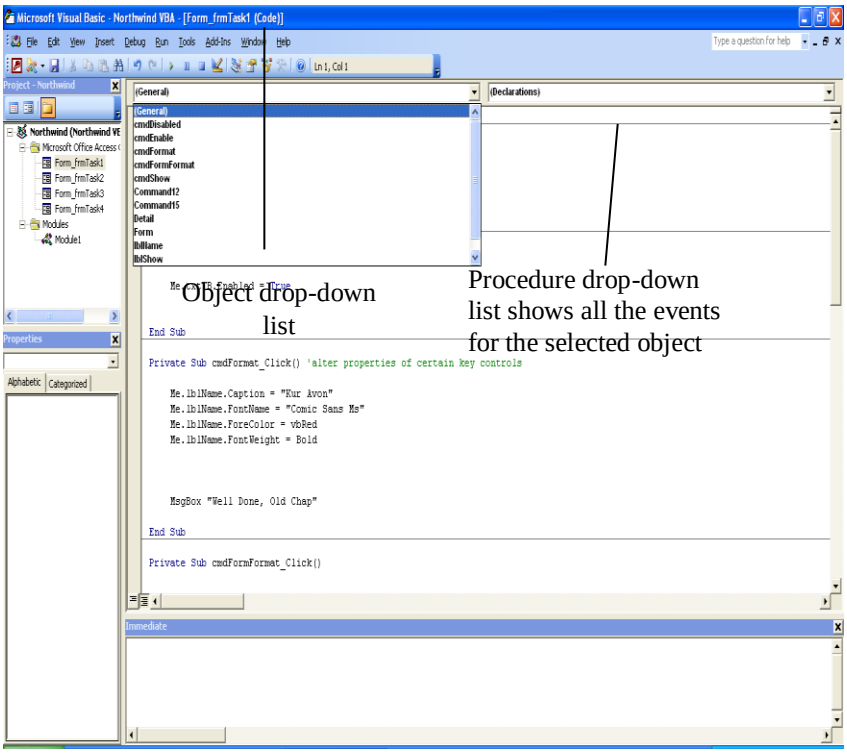
Event procedure names are created automatically. They consist of the object, followed by an underscore and the event name. These names cannot be changed. Event procedures are stored in the class module associated with the object for which they are written.

The syntax of the **Activate Event procedure** is as follows:

```
Private Sub Button_OnClick()
```

Creating An Event Procedure

To create an **Event Procedure**:



- Display the code window for the appropriate class module
- Select the Object from the Object drop-down list
- Select the event from the Procedure drop-down list
- Enter the desired code in the Event Procedure

Unit 4 Using Intrinsic Functions, Variables and Expressions

Defining Expressions and Statements

Any programming language relies on its expressions and the statements that put those expressions to use.

Expressions

An expression is a language element that, alone or in combination represents a value.

The different expression types typical of Visual basic are as follows:

String	Evaluates to a sequence of characters
Numeric	Evaluates to anything that can be interpreted as a number
Date	Evaluates to a date
Boolean	Evaluates to True or False
Object	Evaluates to an object reference

Expressions can be represented by any combination of the following language elements:

Literal	Is the actual value, explicitly stated.
Constant	Represents a value that cannot be changed during the execution of the program. (Eg. vbNo, vbCrLf)
Variable	Represents a value that can be changed during the execution of the program.
Function/Method /Property	Performs a procedure and represents the resulting value. This also includes self-defined functions
Operator	Allows the combination of expression elements +, - , * , / , >, <, =, <>

Statements

A statement is a complete unit of execution that results in an action, declaration or definition.

Statements are entered one per line and cannot span more than one line unless the line continuation character (_) is used.

Statements combine the language's key words with expressions to get things done.

Below are some examples of statements:

```
txtSale.Name = "Quarterly Sales 2006"

Label(1).Caption = ActiveCell.Value

CurrentPrice = CurrentPrice * 1.1
```

How to Declare Variables

A variable is name used to represent a value. Variables are good at representing values likely to change during the procedure. The variable name identifies a unique location in memory where a value may be stored temporarily.

Variables are created by a **Declaration** statement. A variable declaration establishes its name, scope, data type and lifetime.

The syntax for a **Variable declaration** is as follows:

Dim/Public/Private/Static VariableName [As <type>]

Dim EmpName as String

Private StdCounter as Integer

Public TodaysDate As Date

Naming Variables

To declare a variable you give it a name. Visual Basic associates the name with a location in memory where the variable is stored.

Variable names have the following limitations:

- Must start with a letter
- Must NOT have spaces
- May include letters, numbers and underscore characters
- Must not exceed 255 characters in length
- Must not be a reserved word like **True, Range, Selection**

Assigning Values to Variables

An **Assignment** statement is used to set the value of a variable. The variable name is placed to the left of the equal sign, while the right side of the statement can be any expression that evaluates to the appropriate data type.

The syntax for a **Variable declaration** is as follows:

VariableName = expression

StdCounter = StdCounter + 1

SalesTotal = SalesTotal + txtSalesl.Value

Declaring Variables Explicitly

VBA does not require you to explicitly declare your variables. If you don't declare a variable using the **Dim** statement, VBA will automatically declare the variable for you the first time you access the variable. While this may seem like a nice feature, it has two major drawbacks:

- It doesn't ensure that you've spelled a variable name correctly
- It declares new variables as **Variants**, which are slow

Using Dim, Public, Private and Static declaration statements result in **Explicit** variable declarations.

You can force VBA to require explicit declaration be placing the statement **Option Explicit** at the very top of your code module, above any procedure declaration.

With this statement in place, a **Compiler Error - Variable Not Defined** message would appear when you attempt to run the code, and this makes it clear that you have a problem. This way you can fix the problem immediately.

Although this forces you to declare variables, there are many advantages. If you have the wrong spelling for your variable, VBE will tell you. You are always sure that your variables are considered by VBE.

The best thing to do is tell the VBA Editor to include this statement in every new module. See **Setting Code Editor Options** on **Page 21**.

Important Note

When you declare more than one variable on a single line, each variable must be given its own type declaration. The declaration for one variable does not affect the type of any other variable. For example, the declaration:

```
Dim X, Y, Z As Single
```

is **NOT** the same as declaration

```
Dim X As Single, Y As Single, Z As Single
```

It **IS** the same as

```
Dim X As Variant, Y As Variant, Z As Single
```

For clarity, always declare each variable on a separate line of code, each with an explicit data type.

Determining Data Types

When declaring a variable you can specify a data type.

The choice of data type will impact the programs accuracy, efficiency, memory usage and its vulnerability to errors.

Data types determine the following:

- The structure and size of the memory storage unit that will hold the variable
- The kind and range of values the variable can contain. For example in the Integer data type you cannot store other characters or fractions
- The operations that can be performed with the variable such as add or subtract.

Important Info

If data type is omitted or the variable is not declared a generic type called **Variant** is used as default.

Excessive use of the **Variant** data type makes the application slow because Variants consume lots of memory and need greater value and type checks.

Numeric Data Types

Numeric data types provide memory appropriate for storing and working on numbers. You should select the smallest type that will hold the intended data so as to speed up execution and conserve memory.

Numeric operations are performed according to the order of operator precedence:

Operations inside parentheses **( )** are performed first. Access evaluates the operators from left to right.

The following numeric operations are shown in order of precedence and can be used in with numeric data types.

Exponentiation (^)	Raises number to the power of the exponent
Negation (-)	Indicates a negative operand (as in –1)
Divide and Multiply (/ *)	Multiply and divide with floating point result
Modulus (Mod)	Divides two numbers and returns the remainder
Add and Subtract (+ -)	Adds and subtracts operands

String Data Types

The String data type is used to store one or more characters.

The following operands can be used with strings:

Concatenation (&)	Combines two string operands. If an operand is numeric it is first converted to a string-type Variant
Like *LikePattern*	Provides pattern matching strings

VBA supports the following data types:

Data type	Storage size	Range
Boolean	2 bytes	**True** or **False**
Byte	1 byte integer	0 to 255
Integer	2 bytes	-32,768 to 32,767
Long (long integer)	4 byte integer	-2,147,483,648 to 2,147,483,647
Single	4 byte floating point	Approximate range -3.40 x 10^{38} to 3.40 x 10^{38}
Double	8 byte floating point	-1.79769313486231E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values
Currency	8 bytes fixed point	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
String (variable-length)	10 bytes +	0 to approximately 2 billion characters
String (fixed-length)	Length of string	1 to approximately 65,400 characters
Variant (Numeric)	16 bytes	Any numeric value up to the range of a **Double**
Variant (String)	22 bytes +	Same range as for variable-length **String**
Decimal	12 byte (Only used within a Variant)	28 places to the right of the decimal; smallest non-zero number is +/-0.0000000000000000000000000001
Date	8 byte floating point	1 January 100 to 31 December 9999
Object	4 bytes	An address reference to an **Object**

Important Info

For monetary values with up to 4 decimal places use the **Currency** data type.
Single and **Double** data types can be affected by small rounding errors.

A numeric variable of any type may be stored to a numeric variable of another type. The fractional part of a **Single** or **Double** will be rounded off when stored to an Integer type variable.

Programming with Variable Scope

The keywords used to declare variables, Dim, Static, Public or Private, define the scope of the variable. The scope of the variable determines which procedures and modules can reference the variable.

Procedure-Level Variables

These are probably the best known and widely used variables. They are declared (**Dim** or **Static**) inside the Procedure itself. Only the procedure that contains the variable declaration can use it. As soon as the Procedure finishes, the variable is destroyed.

Module-Level Variables

These are variables that are declared (**Dim** or **Private**) outside the Procedure itself in the **Declarations** section of a module.

By default, variables declared with the **Dim** statement in the **Declarations** section are scoped as private. However, by preceding the variable with the **Private** keyword, the scope is obvious in your code.

All variables declared at this level are available to all **Procedures** within the Module. Its value is retained unless the variable is referenced outside its scope, the Workbook closes or the **End Statement** is used.

Public Variables

These variables are declared at the top of any standard **Public** module. **Public** variables are available to all procedures in all modules in a project

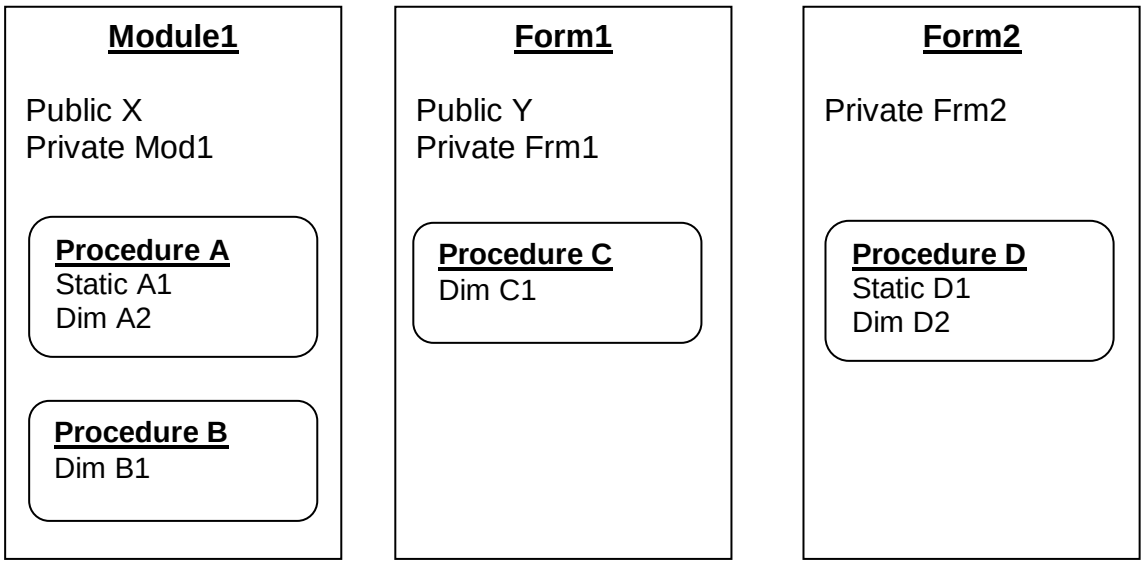
The **Public** keyword can only be used in the **Declarations** section

Public procedures, variables, and constants defined in other than standard or class modules, such as **Form modules** or **Report modules**, are not available to referencing projects, because these modules are private to the project in which they reside.

Variables are processed in the following order:

1. **Local (Dim)**
2. **Module-Level (Private, Dim)**
3. **Public (Public)**

The diagram below illustrates how variables can be accessed across procedures, modules and forms, based on the scope of each variable:



Each of the procedures can only see the variables as follows:

Procedure A can see: A1, A2, Mod1, X, Y

Procedure B can see: B1, Mod1, X, Y

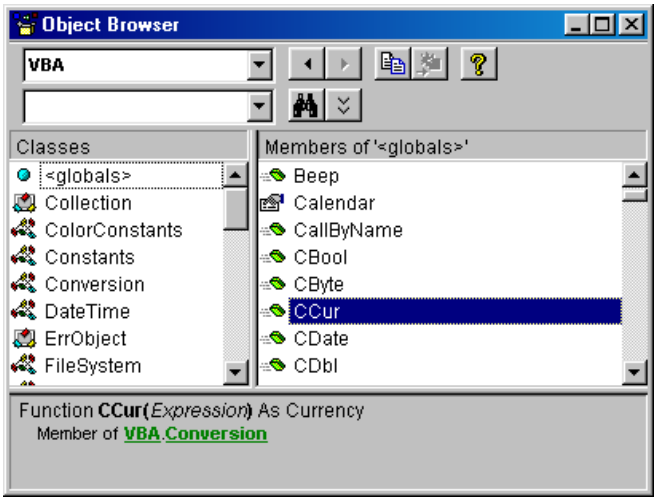
Procedure C can see: C1, Frm1, X, Y

Procedure D can see: D1, D2, Frm2, X, Y

Harnessing Intrinsic Functions

An intrinsic function is similar to a function procedure in that it performs a specific task or calculation and returns a value. There are many intrinsic functions that can be used to manipulate text strings, or dates, covert data or perform calculations.

Intrinsic functions appear as methods in the **Object Browser**. To view and use them:



- Select **VBA** from the **Project/Library** drop down list.
- Select **<globals>** in the **Classes** pane.
- Select the required intrinsic function.

For further help on a particular function, display the **Visual Basic Help** window. On the **Contents** tab:

- Expand Visual Basic Language Reference
- Expand Functions
- Expand the appropriate alphabet range
- Select the desired function.

Defining Constants and Using Intrinsic Constants

A constant is a variable that receives an initial data value that doesn't change during the programs execution. They are useful in situations where a value that is hard to remember appears over and over. The use of constants can make code more readable.

The value of the constant is also set in the declaration statement. Constants are Private by default, unless the Public keyword is used.

The syntax of a **Constant declaration** is as follows:

[Public] [Private] Const ConstantName [<As type>] = <*ConstantExpr*>

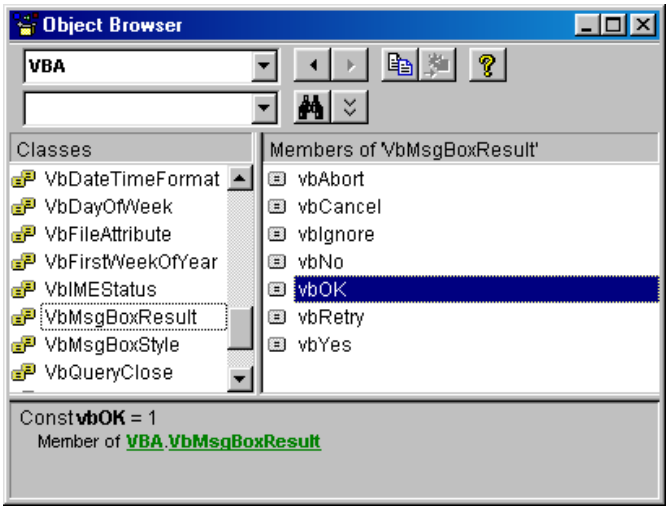
Const conPassMark As String = "C"

```
Public Const conMaxSpeed As Integer = 30
```

Using Intrinsic Constants

VBA has many built-in constants that can be used in expressions. VBA constants begin with the letters ***vb*** while constants belonging to the Access object library begin with ***xl***.

To access Intrinsic constants in the **Object Browser** follow the steps below:



- Select **VBA** from the **Project/Library** drop down list.
- Select the object you want to use in the **Classes** pane e.g. **vbMsgBoxResult**.
- Select the required intrinsic function e.g. **vbOK**

Some useful Visual Basic constants are listed below:

Constant	Equivalent to:	Same as pressing:
vbCr	Carriage Return	Enter
vbTab	Tab character	Tab
vbLf	Soft return and linefeed	Shift + Enter
vbCrLf	Combination of carriage return and linefeed	
vbBack	Backspace character	Backspace
vbNullString	Zero length string	""

For a full list of Visual Basic Constants, search **Help** for **VB Constants** while in the Visual Basic Editor.

Adding Message Boxes

The **MsgBox Function** can be used to display messages on the screen and prompt for a user's response.

The **MsgBox Function** can display a variety of buttons, icons and custom title bar text.

The **MsgBox Function** can be used to return a constant value that represents the button clicked by user.

The **MsgBox Function** syntax is as follows:

MsgBox(prompt[, buttons] [, title] [, helpfile, context])

MyResponse = MsgBox ("Print the new sales report?", 36, _
"Print Sales Report")

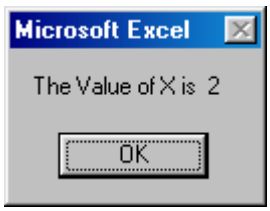
MyResponse = MsgBox ("Print the new sales report?", _
vbYesNo + vbQuestion, "Print Sales Report")

Both **MsgBox Functions** above produce a message box with 2 buttons, a text

message, an icon and a title as shown below:

Another example of using the message box is to return a value:

```
Sub Example()
Dim X As Integer
```




```
X = 2
MsgBox "The Value of X is " & Str(X)
End Sub
```

The **Msgbox** message must be a string (text), hence the **Str() function** is required to convert an integer to a string which is concatenated with the first string using the & operator.

The **MsgBox Function** has the components described below:

prompt Required. It is a string expression displayed as the message in the dialog box. The maximum length of *prompt* is approximately 1024 characters. If *prompt* consists of more than one line, you can separate the lines by concatenating and using carriage return code **vbCrLf.**

buttons Optional. Numeric expression that defines the set of command buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. Can be specified by entering a vbConstant, the actual numeric value of the constant or the sum of constants. If omitted, the default value for *buttons* is 0

title Optional. String expression displayed in the title bar of the dialog box. If you omit **title** "Microsoft Access" is the default title

helpfile Optional. String expression that identifies the Help file to use for the input box. If **helpfile** is provided, **context** must also be provided.

context Optional. Numeric expression that identifies the appropriate topic in the Help file related to the message box

The values and constants for creating buttons are shown below:

Constant	Value	Description
vbOKOnly	0	**OK** button only (default)
vbOKCancel	1	**OK** and **Cancel** buttons
vbAbortRetryIgnore	2	**Abort**, **Retry**, and **Ignore** buttons
vbYesNoCancel	3	**Yes**, **No**, and **Cancel** buttons
vbYesNo	4	**Yes** and **No** buttons
vbRetryCancel	5	**Retry** and **Cancel** buttons

The values for creating icons are shown below:

Constant	Value	Description
vbCritical	16	Display the Stop icon
vbQuestion	32	Display the Question icon
vbExclamation	48	Display the Exclamation icon
vbInformation	64	Display the Information icon

The values for setting the default command button are shown below:

Constant	Value	Description
vbDefaultButton1	0	First button set as default (default)
vbDefaultButton2	256	Second button set as default
vbDefaultButton3	512	Third button set as default
vbDefaultButton4	768	Fourth button set as default

The values for controlling the modality of the message box are shown below:

Constant	Value	Description
vbApplicationModal	0	Application modal message box (default)
vbSystemModal	4096	System modal message box
vbMsgBoxHelpButton	16384	Adds Help button to the message box
VbMsgBoxSetForeground	65536	Specifies the message box window as the foreground window

To display the **OK** and **Cancel** buttons with the **Stop icon** and the second button (Cancel) set as default, the argument would be:

273 (1 + 16 +256).

It is easier to sum the constants than writing the actual values themselves:

vbOKCancel, vbCritical, vbDefaultButton2.

When adding numbers or combining constants, for the button argument, **select only one value**, from each of the listed groups.

Return Values

The **MsgBox Function** returns the value of the button that is clicked. Again this can be referenced by the number or the corresponding constant.

The Return values of the corresponding constants are as follows:

Button Clicked	Constant	Value Returned
OK	vbOK	1
Cancel	vbCancel	2
Abort	vbAbort	3
Retry	vbRetry	4
Ignore	vbIgnore	5
Yes	vbYes	6
No	vbNo	7

The return value is of no interest when the MsgBox only displays the OK button.

In this case just call the **MsgBox Function** with the syntax used to call a sub procedure as shown below:

MsgBox ("You must enter a number", vbOKOnly, "Attention")

Or

MsgBox "You must enter a number"

Using Input Boxes

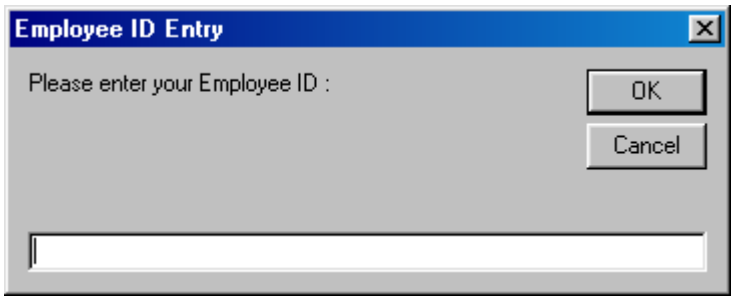
The **InputBox Function** prompts the user for a piece of information and returns it as a string.

The syntax of a **InputBox Function** is as follows:

InputBox (prompt[, title] [, default] [, xpos] [, ypos] [, helpfile, context])

strEmpID = InputBox ("Please enter your Employee ID :", "Employee ID Entry")

In the example the return value of the function is being stored in a variable called



strEmpID.

If **OK** is clicked, the function returns the contents of the text box or a zero-length string, if nothing is entered.

If the user clicks **Cancel**, it returns a zero-length string, which may cause an error in the procedure if a value is required.

The **ImputBox Function** has the components described below:

prompt	Required. String expression displayed in the dialog box. The maximum length of prompt is approximately 1024 characters.
title	Optional. String expression displayed in the title bar of the dialog box. If you omit title "Microsoft Access is the default title.
default	Optional. String expression displayed in the text box as the default response. If you omit default, the text box is displayed empty.
xpos	Optional. Numeric expression that specifies, in **twips**, the horizontal distance of the left edge of the dialog box from the left edge of the screen. If **xpos** is omitted **ypos** must also be omitted.
ypos	Optional. Numeric expression that specifies, in **twips**, the vertical distance of the upper edge of the dialog box from the top of the screen.
helpfile	Optional. String expression that identifies the Help file to use for the Input box. If **helpfile** is provided, **context** must also be provided.
context	Optional. Numeric expression that identifies the appropriate topic in the Help file related to the Input box

A twip is equal to 1/20th of a point.

How To Declare And Use Object Variables

You can also use variables to reference objects in order to work with their properties, methods and events. Any Access object can be represented and accessed using a variable name.

The **Object Variable** syntax is as follows:

Dim/Public/Private/Static VariableName [As <Objecttype>]

Dim Rst As DAO.Recordset

Public FormVar As Form

Assigning values to object variables requires the keyword **Set**:

Set VariableName = Objectname

Set rst = db.openrecordset("Employees")

Set FormVar = "Sales View"

Once an object is assigned to an object variable, the object can be referenced by its variable name. Object variables are used to avoid typing lengthy object references.

Unit 5 Debugging the Code

Understanding Errors

When developing code, problems will always occur. Wrong use of functions, overflow and division by zero are some of the things that will cause an error and not produce the intended results.

Errors are called ***Bugs***. The process of removing bugs is known as ***Debugging***. VBA provides tools to help see how the code is running.

There are three general types of errors:

Syntax Errors

Syntax errors occur when code is entered incorrectly and is typically discovered by the line editor or the compiler.

- **Discovered by Line Editor**: When you move off a line of code in the Code window, the syntax of the line is checked. If an error is detected the whole line turns red by default indicating the line needs to be changed.

- **Discovered by Compiler**: While the line editor checks one line at a time, the compiler checks all the lines in each procedure and all declarations within the project. If **Option Explicit** is set, the compiler also checks that all variables are declared and that all objects have references to the correct methods, properties and events. The compiler also checks that all required statements are present, for example that each **If** has an **End If**. When the compiler finds an error it displays a message box describing the error.

Run-Time Errors

When a program is running and it encounters a line of code that it cannot be executed, a run-time error is generated. These errors occur when a certain condition exists. A condition could run fine 10 times but cause an error on the 11th. When a run-time error occurs, execution is halted a message box appears defining the error.

Logic Errors

Logic errors create unexpected outcomes when a procedure is executed. Unlike syntax or run-time errors the application is not halted and you are not shown the offending line of code. These errors are more difficult to locate and correct.

Minimizing Errors

Here are a few suggestions to help you minimize or make it easier to find errors in your code:

- Add comments to code explaining what a line of code or procedure is meant to do. This is important if other people are going to look at the code.
- Create meaningful variable names. Use prefixes to identify data or object type.
- Any time you use division that contains a variable in the denominator, test the denominator to ensure that it doesn't equal zero
- Force variable declarations with the use of **Option Explicit**. A simple misspelling of a variable name will lead to a logic error, not a run-time error.
- Give procedures names that clearly describe what they do.
- Keep procedures as short as possible, giving it one or two specific tasks to carry out.
- Test procedures with large data sets representing all possible permutations of reasonable or unreasonable data. Make your procedure fail before someone else does.

Using Debugging Tools

VBA's debugging tools are useful for checking and understanding the cause of logic and run-time errors in the code.

The toolbar buttons as they appear left to right are explained below:

Design Mode	Turns design mode off and on.
Run / Continue	Runs code or resumes after a code break
Break	Stops the execution of a program while it's running and switches to Break Mode.
Reset	Clears the execution stack and module level variables and resets the project.
Toggle Breakpoint	Sets or removes a Break Point at the current line.
Step Into	Executes code one statement at a time.
Step Over	Allows selected ode to be stepped over during execution.
Step Out	Executes the remaining lines of a procedure after a break
Locals Window	Displays the value of variables and properties during code execution
Immediate Window	Displays a window where individual lines of code can be executed and variables evaluated.
Watch Window	Displays the value of each expression that is added to a window.
Quick Watch	Displays the current value of the selected expression.
Call Stack	Displays all the currently loaded procedures

Debugging is done when the application is suspended (in **Break Mode**). Everything loaded into memory remains in memory and can be evaluated. A program enters Break mode in one of the following ways

- A code statement generates a run-time error
- A breakpoint is intentionally set on a line of code
- A **Stop** statement is entered within the program code.

Identifying the Value of Expressions

While debugging it is useful to find out the value of variables and expressions while your code is executing.

VBA has the **Locals Window**, **Immediate Window**, **Watch Window** and **Quick Watch**, described in **Using Debugging Tools** on the previous page, which can be used to find the values of expressions

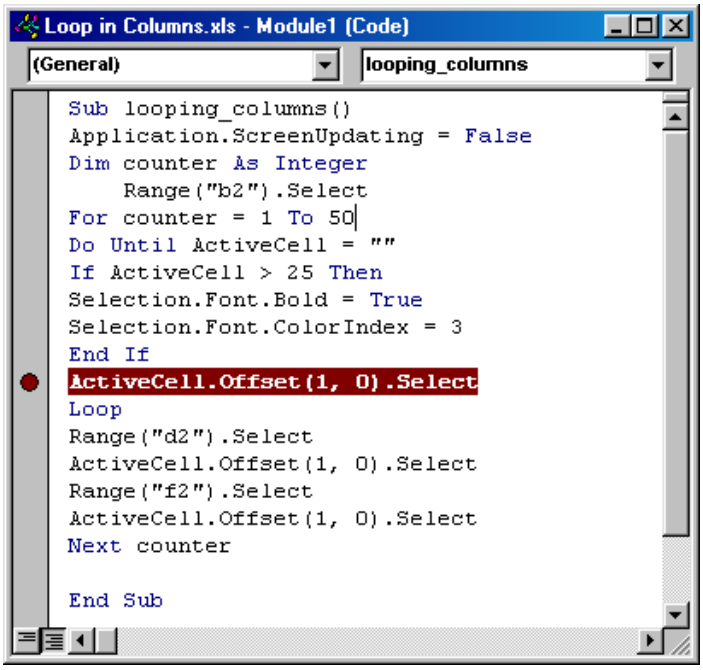
Another quick way of finding out the value of variables and expressions is the **Auto Data Tip** which displays the value of the expression where the mouse is pointing.

Setting Breakpoints

Setting breakpoints allows you to identify the location where you want your program to enter into break mode. The program runs to the line of code and stops. The code window displays and the line of code where the break point is set is highlighted.

When the code is halted, the value of a variable or expression can be checked by holding the mouse pointer over the expression or in the immediate window.

To set a breakpoint open the code window and select the desired procedure:



- Position the insert point on the desired line of code
- Set the breakpoint by clicking **Toggle Breakpoint** on the **Debug toolbar**

OR

- Open the **Debug menu** and select **Toggle Breakpoint**

OR

- Click in the grey area to the left of the line of code

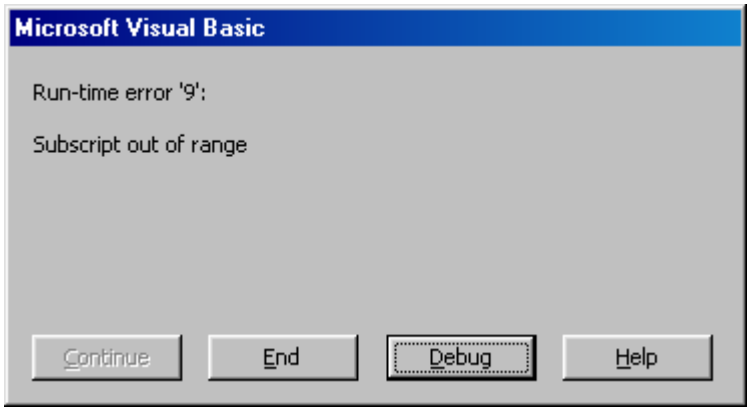
How to Step Through Code

The step tools allow you to step one line at a time through the code to see exactly which statements in your procedure are being executed.

Step Into	F8	Executes code one statement at a time. If the statement calls another procedure execution steps into the called procedure and continues to execute one step at a time.
Step Over	Shift + F8	Executes code one statement at a time. If the statement calls another procedure the procedure is executed without pausing.
Step Out	Ctrl + Shift + F8	Executes the remaining lines of a procedure without pausing.
Run To Cursor	Ctrl + F8	Runs from the current statement to the location of the cursor in the Code window if you are stepping through code.
Set next Statement	Ctrl + F9	Runs the statement of your choice rather than the next statement.
Call Stack	Ctrl + L	Displays all the currently active procedures in the application that have started but are not completed.

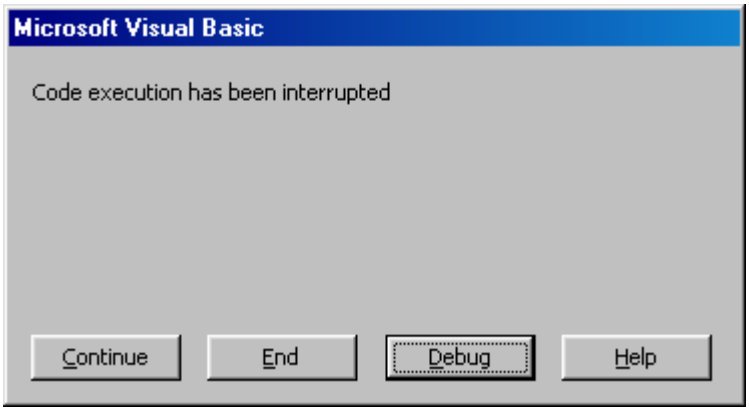
Working With Break Mode During Run Mode

During code execution the program can enter into Break Mode either intentionally or because of a run-time error. When a run-time error occurs a message appears that describes the error.



- Click the **Debug** button to display the code window with the offending line highlighted.

If during the program execution you need to intervene, for example it's stuck in an endless loop, you can do so by pressing **Ctrl + Break** or the **Break button** in the **Visual Basic Editor**.



That action will suspend the program execution and produce the following message:

Unit 6 Handling Errors

Handling errors is another aspect of writing good code. VBA allows you to enter instructions into a procedure that directs the program in case of an error.

Successfully debugging code is more of an art than a science. The best results come from writing understandable and maintainable code and using the available debugging tools. When it comes to successful debugging, there is no substitute for patience, diligence, and a willingness to test relentlessly, using all the tools at your disposal.

Writing good error handlers is a matter of anticipating problems or conditions that are beyond your immediate control and that will prevent your code from executing correctly at run time. Writing a good error handler should be an integral part of the planning and design of a good procedure. It requires a thorough understanding of how the procedure works and how the procedure fits into the overall application. And, writing good procedures is an essential part of building solid Microsoft Office solutions.

Good error handling should keep the program from terminating when an error occurs.

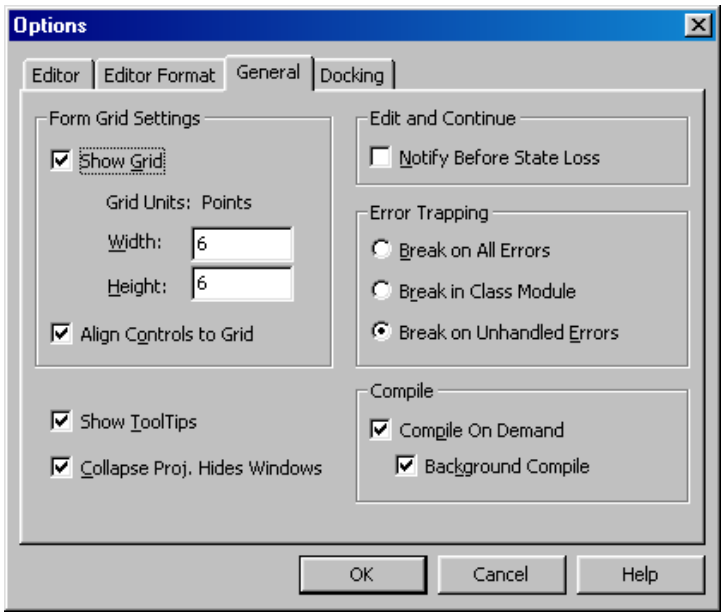
Defining VBA's Error Trapping Options

The error trapping mechanism can be turned on, off or otherwise modified while developing a project.

To set the **Error Handling** options:

- Open the **Tools** menu
- Select **Options**

The **Options** dialog box appears.



The **Error Trapping options** are explained below:

Break on All Errors	Causes program to enter Break mode and display an error message regardless of whether you have written code to handle the error. This option turns the error handling mechanism off and should be used for debugging only
Break in Class Module	Causes program to enter Break mode and display an error message when an unhandled error occurs within a procedure of a class module such as a User Form. If the Debug button is clicked in the error message window, the Code window will display the line of code that generated the error highlighted. Should be used for debugging only.
Break on Unhandleded Errors	Causes the program to enter Break mode and display a message when an unhandled error occurs. **This is the setting that should be selected before distributing your application.**

For a list of trappable errors in Access search **Help** for **Trappable Errors Constants** while in the Visual Basic Editor.

A list of the error numbers and their descriptions appears.

Capturing Errors with the On Error Statement

In a procedure, you enable an error trap with an **On Error** statement. If an error is generated after this statement in encountered, the Error handler takes over and passes control to what the **On Error** statement specifies.

The Error-Handling syntax is as follows:

On Error <*branch instruction*>

On Error GoTo ErrorHandler

On Error Resume Next

Once an On Error statement has trapped an error, the error needs to be handled. Below are the 3 basic styles that VBA uses for handling errors:

Write an Error handler	This uses the **On Error GoTo** statement. It would include statements to handle one or more errors for the procedure.
Ignore the Error	If the error is inconsequential, use the **On Error Resume Next** statement to both trap and handle the error. The program continues on the next line of code.
Use in-line error handling	Use the **On Error Resume Next** statement to trap the error. Then enter code to check for errors immediately following any statements expected to generate errors.

On Error GoTo 0

This statement disables the error-handling for the procedure at least until another **On Error statement** is encountered. This is an alternative to changing the **Error Trapping** settings to **Break on All Errors** as it only affects the procedure it is in. Once the issue is resolved remove the statement from the procedure.

Error trapping is defined on a procedure-by-procedure basis. VBA does not allow you to specify a global error trap.

Determining the Err Object

When an error occurs, VBA uses the **Err** object to store information about that error. The **Err** object can only contain information about one error at a time

The properties of the **Err** object contain information such as the Error **Number**, **Description**, and **Source**.

The **Err** object's **Raise** method is used to generate errors, and its **Clear** method is used to remove any existing error information.

Using the Raise methods to force an error can help in error testing routines.

The following statement generates a "Division By Zero" error message:

Err.Raise 11

Coding An Error-Handling Routine

The On Error Go To statement is used to branch to a block of code within the same procedure which handles errors. This block is known as the error-handling routine and is identified by a line label.

The routine is always stored at the bottom of the procedure, preceded by an **Exit** statement that prevents the routine from being executed unless an error has occurred.

Common line labels used to identify an Error-handling routine are "ErrorHandler" and "EH". You can use one of these or create a personal one to handle all your error-handling routines.

Line labels only have to be unique within the procedure.

The benefit of using this style is that all the error-handling logic is at the bottom rather than being mixed up with the main logic of the procedure making the procedure easier to read and understand.

The example below illustrates a error-handling routine for a sub procedure:

```
Sub RunFormula()

On Error GoTo ErrorHandler

Dim A As Double
Dim B As Double

A = InputBox("Type in the value for A")
B = InputBox("Type in the value for B")

MsgBox A / B

Exit Sub

ErrorHandler:

If Err.Number = 11 Then
   B = InputBox(Err.Description & " is not allowed.  Enter a non-zero number.")
   Resume
Else
   MsgBox "Unexpected Error.  Type " & Err.Description
End If

End Sub
```

When an execution has passed into an error routine the following list shows how to specify which code to be used next:

Resume	Execution continues on the same line within the procedure that caused the error.
Resume Next	Execution continues on the line within the procedure that follows the line that caused the error.
Resume <*Line Label*>	Execution continues on the line identified by the line label. This usually points to another routine within the procedure that performs a "clean-up" be releasing variables and deleting temporary files.
End Sub / End Function	Used to exit the procedure normally by reaching the End Sub or end Function command
Exit Sub / Exit Function	Immediately exits the procedure in which it appears. Execution continues with the statement following the statement that called the procedure.

Using Inline Error Handling

Using this method you place the code to handle errors directly into the body of the procedure, rather than placing it at the end of the routine.

To do this, place the **On Error Resume Next** statement into the procedure. The error handling code is then placed immediately after the line where the code is expected to cause error. This method may be simpler to use in very long procedures where two or more errors are anticipated.

```
Sub ProcFileOpen()

On Error Resume Next

Open "C:\My Documents\Sales2006.xls" For Input As #1
Select Case Err
   Case 53
      MsgBox "File not found:  C:\My Documents\Sales2006.xls"
    Case 55
      MsgBox "File in use:  C:\My Documents\Sales2006.xls"
    Case Else
      MsgBox "Err Number: " & Err.Number & vbLf & _
      "Error Descriptoion: " & Err.Description
End Select

Err.Clear

End Sub
```

Unit 7 Managing Program Execution

Defining Control-Of-Flow structures

When a procedure runs, the code executes from top to bottom in the order that it appears. Only the simplest of programs execute in this manner. Most programs incorporate logic to control which lines of code to execute.

The **Control-Of-Flow** structures described below provide this logic:

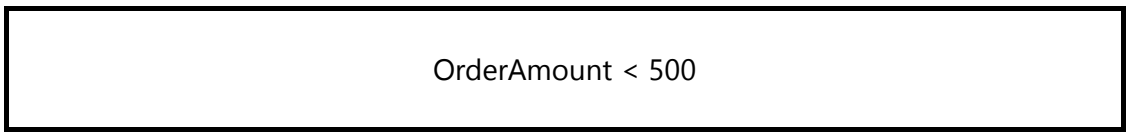
Sequential	Each line of code is executed in order from top to bottom.
Unconditional Branching	A statement that directs the flow of program execution to another location in the program without condition. Calling a **Function**, a **Sub** or using the **GoTo** statement are examples of unconditional branching
Conditional Branching	The code to be executed is based on the outcome of a **Boolean** expression. Decision structures like **If** and **Select Case** are used to implement conditional branching.
Looping	A block of code executed repeatedly as long as a certain condition exists. The **For…Next** and the **Do..Loop** are examples of looping structures
Halt Statements	Commands used to stop code execution. The **Stop** command stops execution but retains variables in memory. The **End** command terminates the application.

Using Boolean Expressions

A **Boolean** expression returns a True or False value. Many **Boolean** expressions take the form of two expressions either side of a comparison operator. If the result is true the condition is met and control is passed to the code to be executed.

Here are some examples of **Boolean** expressions:

```
Firstname = "Alan"

UnitPrice > 1.60
```


OrderAmount < 500

The following comparison operators are used in **Boolean** expressions:

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
=	Equal to
<>	Not equal to
Is	Compares object variables
Like	Compares string expressions

When testing for more than one condition **Boolean** expressions can be joined with a **Logical Operator**.

The following is a list of **Logical Operators**:

And	Each expression must be True for the condition to be true.
Or	One of the expressions must be True for the condition to be true.
Not	The expression must be False for the condition to be true.

The following are examples of multiple conditions joined by logical operator:

```
UnitPrice > 1.60 AND OrderAmount > 1000

DateJoined <= 2004 OR DeptName = "Sales"
```

A null expression will be treated as a false expression.

Using the If…End If Decision Structures

If…End If is used to execute one or more statements depending upon a text condition. There are four forms of the **If** construct.

The first contains the condition and statement to be executed in the same line:

If <*condition*> Then <*statement*>

```
If OrderAmount >1000 Then Discount = "Yes"
```

The block form is used when several statements are to be executed based on result of the test condition:

If <*condition*> Then
**   <*statement block*>**
End If

```
If Country = "England" Then
    Account = "Domestic"
    TransportCost = 10.00
End If
```

Like the **If…Then** structure the **If…Then…Else** structure passes control to the statement block that follows the **Then** keyword when the condition is **True** and passes control to the statement block that follows the **Else** keyword when the condition is **False**.

If <*condition*> Then
**   <*statement block*>**
Else
**   <*statement block*>**
End If

```
If Country = "England" Then
    Account = "Domestic"
    TransportCost = 10.00
Else
    Account = "Foreign"
    TransportCost = 40.00
```

```
End If
```

By modifying the basic structure and inserting **ElseIf** statements, an **If…Then…Else** block that tests multiple conditions is created. The conditions are tested in the order of appearance until a condition is true.

If a true condition is found, the statement block following the condition is performed; execution then continues with the first line of code following the **End If** statement. If no condition is true, execution will continue with the **End If** statement. An optional **Else** clause at the end of the block will catch the cases that do not meet any of the conditions.

If <*condition_1*> Then
**   <*statementBlock1*>**
[ElseIf <*condition_2*> Then
**   [<*StatementBlock2*>]]**
[ElseIf <*condition_3*> Then
**   [<*StatementBlock3*>]]**
[ElseIf <*condition_N*> Then
**   [<*StatementBlockN*>]]**
End If

```
If Country = "England" Then
    Account = "Domestic"
    TransportCost = 10.00
ElseIf Country = "Wales" Then
    Account = "Domestic"
    TransportCost = 20.00
ElseIf Country = "Scotland" Then
    Account = "Domestic"
    TransportCost = 25.00
ElseIf Country = "Northern Ireland" Then
    Account = "Domestic"
    TransportCost = 30.00
Else
    Account = "Foreign"
    TransportCost = 40.00

End If
```

Using the Select Case…End Select Structure

The **Select Case** statement is often used in place of the complex **If** statement. The advantage of using this style is that your code will be more readable and efficient. The downside is that it is only useful if compared against just one value.

The **Select Case** structure contains the test expression in the first line of the block. Each **Case** statement in the structure then compares against the test expression.

The syntax of the **Select Case** structure, followed by two examples is shown below:

```
Select Case <TestExpression>
Case <Expression_1>
        <StatementBlock1>
Case <Expression_2>
        <StatementBlock2>
Case <Expression_3>
        <StatementBlock3>
Case <Expression_N>
        <StatementBlockN>
End Select
```

```
Select Case Country
    Case "England"
            Account = "Domestic"
            TransportCost = 10.00
    Case "Wales"
            Account = "Domestic"
            TransportCost = 20.00
    Case "Scotland"
            Account = "Domestic"
            TransportCost = 25.00
    Case "Northern Ireland"
            Account = "Domestic"
            TransportCost = 30.00
    Case Else
            Account = "Foreign"
            TransportCost = 40.00
End Select
```

```
Select Case TestScore
    Case 0 To 50
            Result = "Below Average"
    Case 51 To 70
            Result = "Good"
    Case Is > 70
```

```
                Result = "Excellent"
        Case Else
                Result = "Irregular Test Score"
End Select
```

Using the Do…Loop Structure

The **Do…Loop** structure controls the repetitive execution of the code based upon a test of a condition. There are two variations of the structure: **Do While** and **Do Until**.

The **Do While** structure executes the code as long as the condition is true.
The **Do Until** structure executes the code up to the point where the condition becomes true or as long as the condition is false. The condition is any expression that can be evaluated to true or false.

The **Exit Do** is optional and can be used to quit the **Do** statement and resume execution with the statement following the Loop. Multiple **Exit Do** statements can be placed anywhere within the Loop construct.

The following syntax is used to perform the statement block zero or more times:

```
Do While <condition>
    <statement block>
[Exit Do]
Loop

Do Until <condition>
    <statement block>
[Exit Do]
Loop
```

```
Do While ActiveCell.Value <> ""
    ActiveCell.Value = ActiveCell.Value *1.25
    ActiveCell.Offset(1).Select
Loop
```

To perform the statement block at least once, use one of the following:

```
Do
    <statement block>
[Exit Do]
Loop While <condition>

Do
    <statement block>
[Exit Do]
Loop Until <condition>
```

```
Do
    Count = Count +1
Loop Until Count = NoStudents
```

Using the For…Next Structure

The **For…Next** structure executes a block of statements a specific number of times using a counter that increases or decreases values. Beginning with the start value, the counter is increased or decreased by the increment. The default increment is 1. Specify an increment of -1 to count backwards.

The **Exit For** statement is optional and can be used to quit the **For** construct and resume execution with the statement following the **Next**.

Below is the syntax of the **For…Next** statement:

```
For <counter> = <start> To <end> [Step <increment>
        <statement block>
        [Exit For]
Next [<counter>]

Dim MyIndex as Integer
        For MyIndex = 1 To NoRows
            Cells (MyIndex,4).Select
            Total = Total + Cells (NoRows,4).Value
Next MyIndex
```

Using the For Each…Next Structure

The **For Each…Next** structure is used primarily to loop through a collection of objects. With each loop it stores a reference to a given object within the collection to a variable. The variable can be used by the code to access the object's properties. By default it will loop through ALL the objects in a collection.

The **Exit For** statement is optional and can be used to quit the **For Each** construct and resume execution with the statement following the **Next**.

Below is the syntax of the **For Each…Next** statement:

```
For Each <element> in <CollectionReference>
        <statement block>
        [Exit For]
Next [<element>]

Dim rst As dao.recordset
```

```
For Each rst In Application.recordsets
    rst.delete
Next rst
```

Guidelines for Use of Control-Of-Flow Structures

Use the following as a guide in choosing the appropriate **Decision** structure:

Use	To
If…Then Or **If…Then…End If**	Execute one statement based on the result of one condition
If…Then…End If	Execute a block of statements based on the result of one condition
If…Then…Else…End If	Execute 1 of 2 statement blocks based on the result of one condition
Select Case…End Select	Execute 1 of 2 or more statement blocks based on 2 or more conditions, with all conditions evaluated against 1 expression.
If…Then…ElseIf…End If	Evaluate 1 of 2 or more statement blocks based on 2 or more conditions, with conditions evaluated against 2 or more expressions.

Use the following as a guide in choosing the appropriate **Looping** structure:

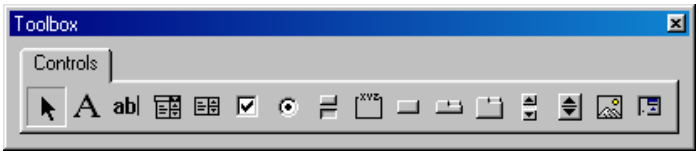
Use	To
For…Next	Repeat a statement block a specific number of times. The number is known or calculated at the beginning of the loop and doesn't change.
For…Each	Repeat a statement block for each element in a collection or array.
For…Next	Repeat a statement block while working through a list when the number of list items is known or is calculated beforehand.
Do…Loop	Repeat a statement block while working through a list when the number of list items is not known or are likely to change.

Do…Loop	Repeat a statement block while a condition is met.

Unit 8 Harnessing Forms and Controls

Defining Forms

Dialog boxes are used in applications to interface with the user. VBA allows you to



create custom dialog boxes that can display information or retrieve information from the user as required. These are known as **Forms** or just **Forms**.

A Form serves as a container for control objects, such as labels, command buttons, combo boxes, etc. These controls depend on the kind of functionality you want in the form. When a new Form is added to the project, the Form window appears with a blank form, together with a toolbox containing the available controls. Controls are added by dragging icons from the toolbox to the Form. The new control appears on the form with 8 handles that can be used to resize the control. The grid dots on the form help align the controls on the form.

Utilising the Toolbox

While working on a form the toolbox is displayed but becomes hidden when another window in the Visual Basic Editor is selected. Controls are added to forms to build a desired interface and add functionality.

The default set of controls, from left to right, on the above toolbox are described below:

Select Objects	Makes the mouse behave as a pointer for selecting a control on a form.
Label	Creates a box for static text
Text Box	Creates a box for text input or display.
Combo Box	Creates the combination of a drop-down list and textbox. The user can select an option or type the choice.
List Box	Creates a scrollable list of choices
Check Box	Creates a logical check box
Option Button	Creates an option button that allows exclusive choice from a set of options.

Toggle Button	Creates a toggle button that when selected indicates a **Yes**, **True** or **On** status.
Frame	Creates a visual or functional border.

Command Button	Creates a standard command button.
Tab Strip	Creates a collection of tabs that can be used to display different sets of similar information.
MultiPage	Creates a collection of pages. Unlike the Tab Strip each page can have a unique layout.
Scroll Bar	Creates a tool that returns a value of for a different control according to the position of the scroll box on the scroll bar
Spin Button	Creates a tool that increments numbers.
Image	Creates an area to display a graphic image.
RefEdit	Displays the address of a range of cells selected on one or more worksheets.

Double-click a toolbox icon and it remains selected allowing multiple controls to be drawn.

Using Form Properties, Events and Methods

Every **Form** has its own set of properties, events and methods. Properties can be set in both the Properties window and through code in the Code window.

Properties

All forms share the same basic set of properties. Initially every form is the same. As you change the form visually, in the Form window, you are also changing its properties. For example if you resize a form window, you change the Height and Width properties.

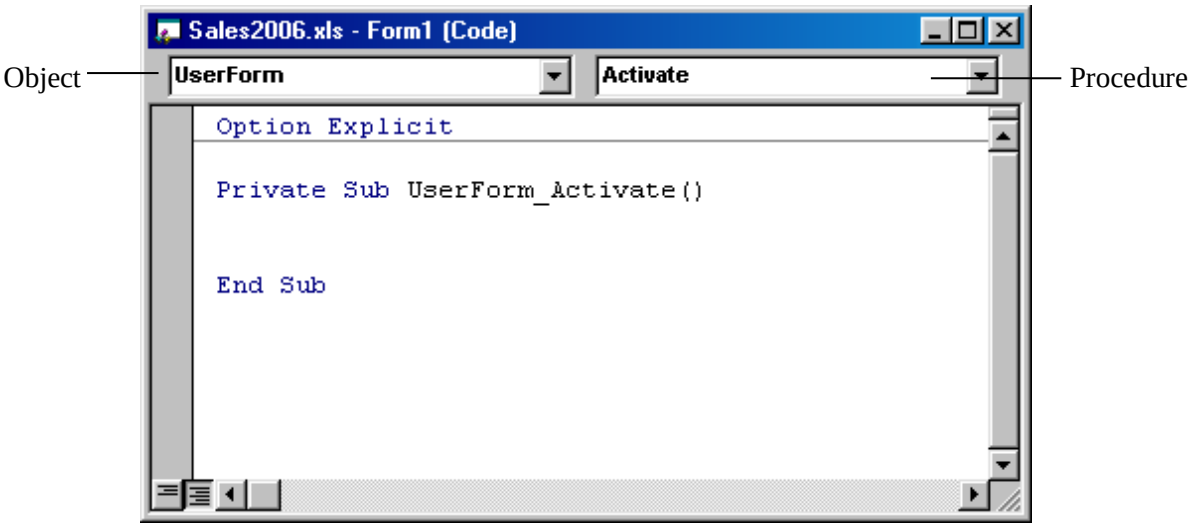
The following list describes the more commonly used properties of a Form:

Property	Description
BackColor	Sets the background colour of a form.
BorderStyle	Sets the border style for the form.
Caption	Sets the form's title in the title bar.
Enabled	Determines whether the form can respond to user-generated events.
Height	Sets the height of the form.
HelpCOntextID	Associates a context-sensitive Help topic with a form.
MousePointer	Sets the shape of the mouse pointer when the mouse is positioned over the form.
Picture	Specifies picture to display in the form.
StartUpPosition	Sets where on the screen the form will be displayed.
Width	Sets the width of the form.

Events

All **Forms** share a set of events they recognize and to which they respond by executing a procedure. You create the code to execute for a form event the same way as you create other event procedures:

- Display the code window for the form
- Select the **Form** object
- Select the event from the **Procedure** list.



Methods

Forms also share methods that can be used to execute built-in procedures. Methods are normally used to perform an action in the form.

The three most useful methods are explained below:

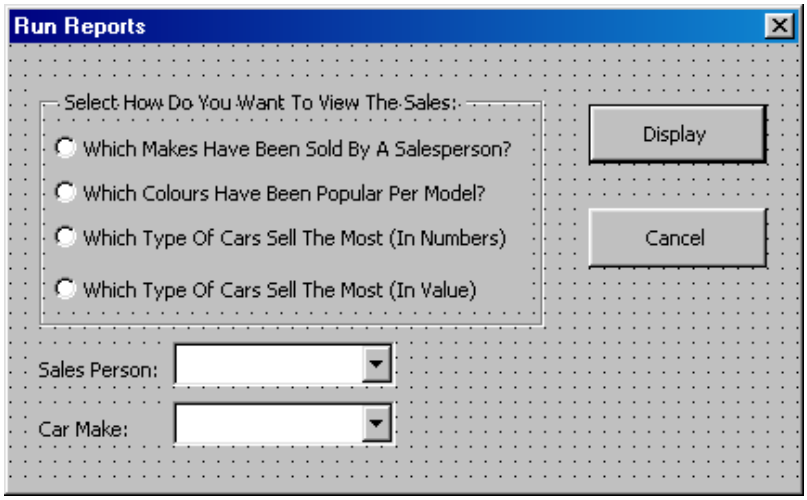
Show	Displays the form; can be used to load a form if not already loaded.
Hide	Hides the form without unloading it from memory.

Use the keyword **Me** in the Form's code module instead of its name to refer to the active form and access its properties and methods.

Understanding Controls

A control is an object placed on a form to enable user interaction. Some controls accept user input while others display output. Like all other objects controls can be defined by their properties, methods and events.

Below is an example of a form containing commonly used controls:



Control properties can be viewed and assigned manually via the Properties window. While each type of control is unique many share similar attributes.

The following list contains properties that are common among several controls:

Property	Description
ControlTipText	Specifies a string to be displayed when the mouse pointer is paused over the control
Enabled	Determines if the user can access the control.
Font	Sets the control text type and size.
Height	Sets the height of the control
MousePointer	Sets the shape of the mouse pointer when the mouse is positioned over the object
TabIndex	Determines the order in which the user tabs through the controls on a form.
TabStop	Determines whether a control can be accessed using the

	tab key.
Visible	Determines if a control is visible
Width	Sets the width of a control.

All controls have a default property that can be referred by simply referencing the name of the control. In one example the **Caption** property is the default property of the **Label** control.

This makes the two statements below equivalent:

```
Label1 = "Salary"
Label1.Caption = "Salary"
```

As with forms many controls respond to system events.

The following are the more common events that controls can detect and react to:

Click	Occurs when the user clicks the mouse button while the pointer is on the control
GotFocus	Occurs when a control receives focus
LostFocus	Occurs when a control loses focus
MouseMove	Occurs when a user moves the mouse pointer over a control.

Naming Conventions

It's a good practice to use a prefix that identifies the control type when you assign a name to the control.

Below is a list of several control object name prefix conventions:

Object	Prefix
Check box	chk
Combo box	cbo
Command button	cmd
Frame	fra
Image	img
Label	lbl
List box	lst

Option button	opt
Text box	txt

Events List

Name	Description
Activate	The Activate event occurs when a form receives the focus and becomes the active window.
AfterDelConfirm	The **AfterDelConfirm** event occurs after the user confirms the deletions and the records are actually deleted or when the deletions are canceled.
AfterFinalRender	Occurs after all elements in the specified PivotChart view have been rendered.
AfterInsert	The **AfterInsert** event occurs after a new record is added.
AfterLayout	Occurs after all charts in the specfied PivotChart view have been laid out, but before they have been rendered.
AfterRender	Occurs after the object represented by the *chartObject* argument has been rendered.
AfterUpdate	The **AfterUpdate** event occurs after changed data in a control or record is updated.
ApplyFilter	Occurs when a filter is applied to a form.
BeforeDelConfirm	The **BeforeDelConfirm** event occurs after the user deletes to the buffer one or more records, but before Microsoft Access displays a dialog box asking the user to confirm the deletions.
BeforeInsert	The BeforeInsert event occurs when the user types the first character in a new record, but before the record is actually created.
BeforeQuery	Occurs when the specified PivotTable view queries its data source.
BeforeRender	Occurs before any object in the specified PivotChart view has been rendered.
BeforeScreenTip	Occurs before a ScreenTip is displayed for an element in a PivotChart view or

	PivotTable view.
BeforeUpdate	The **BeforeUpdate** event occurs before changed data in a control or record is updated.
Click	The **Click** event occurs when the user presses and then releases a mouse button over an object.
Close	The **Close** event occurs when a form is closed and removed from the screen.
CommandBeforeExecute	Occurs before a specified command is executed. Use this event when you want to impose certain restrictions before a particular command is executed.
CommandChecked	Occurs when the specified Microsoft Office Web Component determines whether the specified command is checked.
CommandEnabled	Occurs when the specified Microsoft Office Web Component determines whether the specified command is enabled.
CommandExecute	Occurs after the specified command is executed. Use this event when you want to execute a set of commands after a particular command is executed.
Current	Occurs when the focus moves to a record, making it the current record, or when the form is refreshed or requeried.
DataChange	Occurs when certain properties are changed or when certain methods are executed in the specified PivotTable view.
DataSetChange	Occurs whenever the specified PivotTable view is data-bound and the data set changes — for example, when a filter operation takes place. This event also occurs when initial data is available from the data source.
DblClick	The **DblClick** event occurs when the user presses and releases the left mouse button twice over an object within the double-click time limit of the system.
Deactivate	The **Deactivate** event occurs when a form loses the focus to a Table, Query, Form, Report, Macro, or Module window, or to the Database window.
Delete	Occurs when the user performs some action, such as pressing the DEL key, to delete a record, but before the record is actually deleted.
Dirty	The Dirty event occurs when the contents of the specified control changes.
Error	The Error event occurs when a run-time error is produced in Microsoft Access when a form has the focus.

Filter	Occurs when the user opens a filter window by clicking **Filter by Form**, **Advanced Filter/Sort**, or **Server Filter By Form**.
GotFocus	The **GotFocus** event occurs when the specified object receives the focus.
KeyDown	The **KeyDown** event occurs when the user presses a key while a form or control has the focus. This event also occurs if you send a keystroke to a form or control by using the SendKeys action in a macro or the **SendKeys** statement in Visual Basic.
KeyPress	The **KeyPress** event occurs when the user presses and releases a key or key combination that corresponds to an ANSI code while a form or control has the focus. This event also occurs if you send an ANSI keystroke to a form or control by using the SendKeys action in a macro or the **SendKeys**statement in Visual Basic.
KeyUp	The **KeyUp** event occurs when the user releases a key while a form or control has the focus. This event also occurs if you send a keystroke to a form or control by using the SendKeys action in a macro or the **SendKeys** statement in Visual Basic.
Load	Occurs when a form is opened and its records are displayed.
LostFocus	The **LostFocus** event occurs when the specified object loses the focus.
MouseDown	The **MouseDown** event occurs when the user presses a mouse button.
MouseMove	The **MouseMove** event occurs when the user moves the mouse.
MouseUp	The **MouseUp** event occurs when the user releases a mouse button.
MouseWheel	Occurs when the user rolls the mouse wheel in Form View, Split Form View, Datasheet View, Layout View, PivotChart View, or PivotTable View.
OnConnect	Occurs when the specified PivotTable view connects to a data source.
OnDisconnect	Occurs when the specified PivotTable view disconnects from a data source.
Open	The **Open** event occurs when a form is opened, but before the first record is displayed.
PivotTableChange	Occurs whenever the specified PivotTable view field, field set, or total is added or deleted.
Query	Occurs whenever the specified PivotTable view query becomes necessary. The query may not occur immediately; it may be delayed until the new data is displayed.
Resize	The **Resize** event occurs when a form is opened and whenever the size of a form

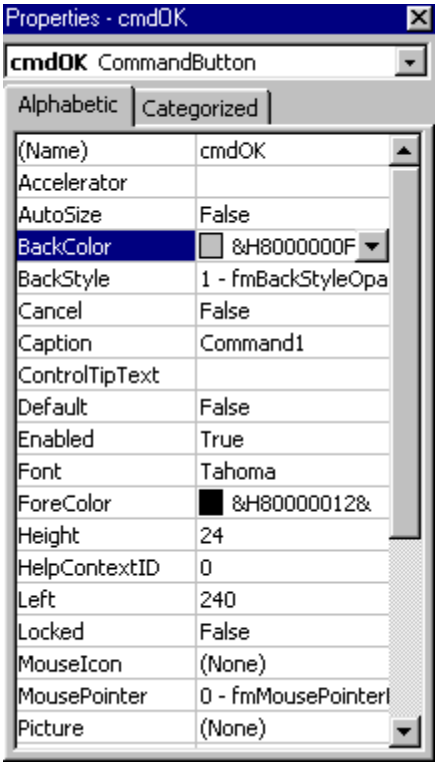
	changes.
SelectionChange	Occurs whenever the user makes a new selection in a PivotChart view or PivotTable view.
Timer	The **Timer** event occurs for a form at regular intervals as specified by the form's **TimerInterval** property.
Undo	Occurs when the user undoes a change.
Unload	The **Unload** event occurs after a form is closed but before it's removed from the screen. When the form is reloaded, Microsoft Access redisplays the form and reinitializes the contents of all its controls.
ViewChange	Occurs whenever the specified PivotChart view or PivotTable view is redrawn.

Setting Control Properties in the Properties Window

Each control has a set of properties that can be set in the design environment using the Properties window. Categories for the property window vary per object.

Frequently used categories are behaviour, font, and position.

To set **Control Properties** in the Properties Window:



- Display the **Properties Window**
- Click the **Alphabetic** tab to display properties in alphabetic order **OR**
- Click the **Categorized** tab to display properties by category

To change a property setting:

- Select the desired control in the Form window or from the drop down list in the Properties window
- Scroll to the desired property and use the appropriate method to change the setting in the value column.

Using the Label Control

The **Label** control is used to display text on a form that cannot be modified by the user.

It can be modified in the procedure by using the **Caption** property.

Below are some unique properties of the **Label** control:

Property	Description
TextAlign	Determines the alignment of the text inside the label.
AutoSize	Determines if the dimensions of the label will automatically resize to fit the caption.
Caption	Sets the displayed text of the field.
WordWrap	Determines if a label expands horizontally or vertically as text is added. Used in conjunction with the **AutoSize** property.

Using the Text Box Control

The **Text Box** control allows the user to add or edit text. Both string and numeric values can be stored in the Text property of the control.

Below are some important properties of the **Text Box** control:

Property	Description
MaxLength	Specifies the maximum number of characters that can be typed into a text box. The default is 0 which indicates no limit.
MultiLine	Indicates if a box can contain more than one line.
ScrollBars	Determines if a multi-line text box has horizontal and/or vertical scroll bars.
Text	Contains the string displayed in the text box.

Using the Command Button Control

Command buttons are used to get feedback from the user. Command buttons are among the most important controls for initiating event procedures.

The most used event associated with the **Command Button** is the **Click** event.

Below are two unique properties of the **Command button** control:

Property	Description
Cancel	Allows the **Esc** key to "click" a command button. This property can only be set for one command button per form.
Default	Allows the **Enter** key to "click" a command button. This property can only be set for one command button per form.

Using the Combo Box Control

The **Combo Box** control allows you to display a list of items in a drop-down list box. The user can select a choice from the list or type an entry.

The items displayed on the list can be added in code using the **AddItem** method.

Below are some important properties of the **Combo Box** control:

Property	Description
ListRows	Sets the number of rows that will display in the list.
MatchRequired	Determines whether the user can enter a value that is not on the list.
Text	Returns or sets the text of the selected row on the list.

Some important methods that belong to the **Combo Box** are explained below:

AddItem *item_name, index*	Adds the specific item to the bottom of the list. If the index number is specified after the item name its added to that position on the table
RemoveItem *index*	Removes the item referred to by the index

	number.
Clear	Clears the entire list.

Using the Frame Control

The **Frame** control is used to group a set of controls either functionally or logically within an area of a **Form**. Buttons placed within a frame are usually related logically so setting the value of one affects the values of others in the group.

Option buttons is a frame are mutually exclusive, which means when one is set to true the others will be set to false.

Using Option Button Controls

An **Option Button** control displays a button that can be set to on or off. Option buttons are typically presented within a group in which one button may be selected at a time.

The **Value** property of the button indicates the on and off state.

Using Control Appearance

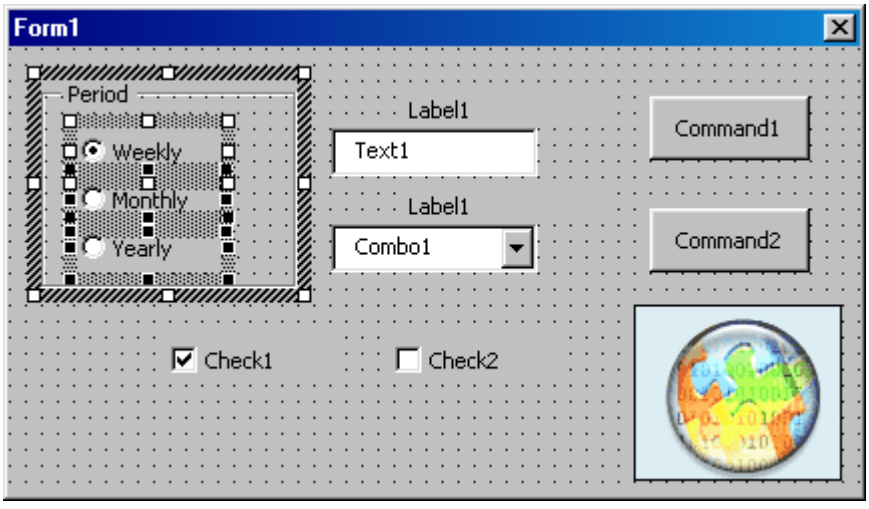
The Form toolbar provides several tools that are used to manipulate the appearance of the controls on the form.

Many of the tools on the Form toolbar require the user to select multiple controls. To do this:

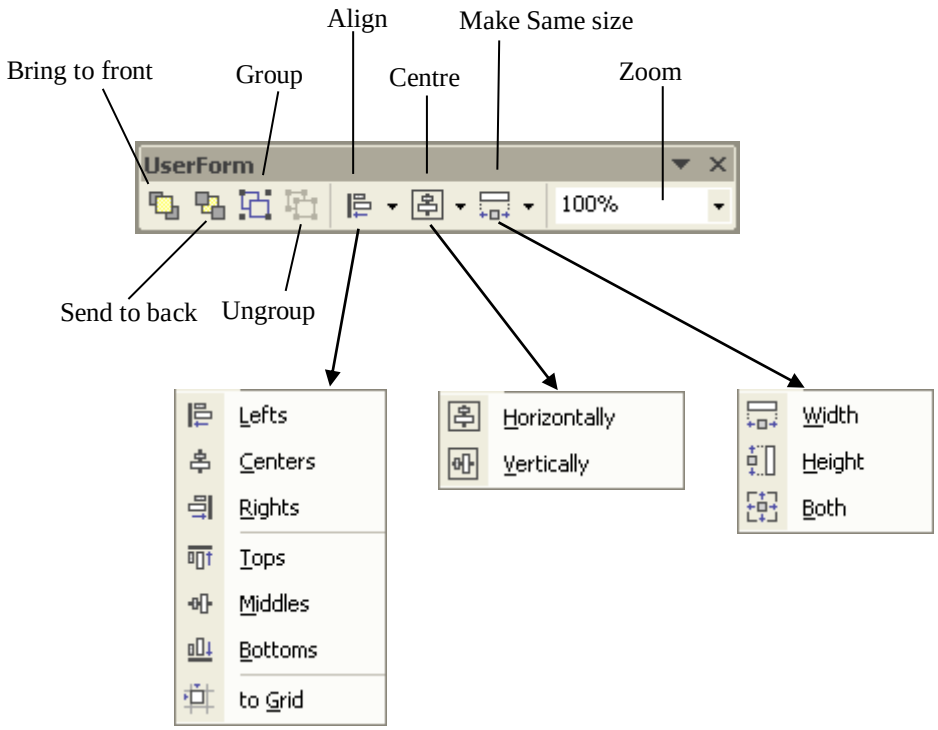
- Click the first control
- Hold down the **Shift** key
- Click any additional controls

Controls will be aligned or sized according to the first control selected. The first control selected is identified by its white selection handles.

Below is an illustration of a Form with multiple controls selected:

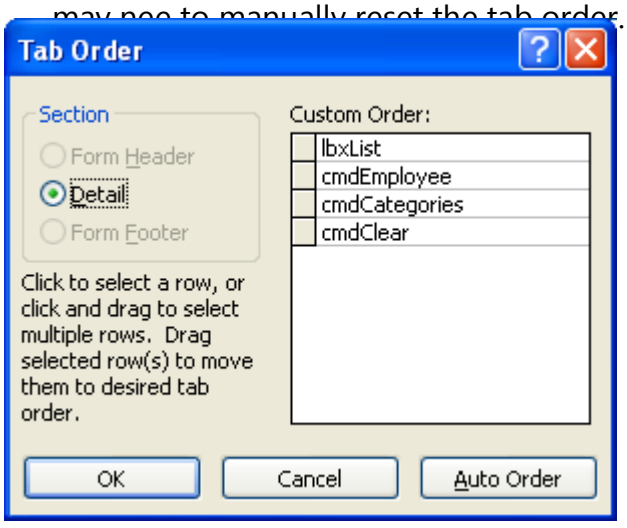

Form1
Period
Weekly
Monthly
Yearly
Label1
Text1
Label1
Combo1
Command1
Command2
Check1
Check2

Below is an illustration of the **Form** toolbar together with the options for **Align**, **Centre** and **Make Same Size**.



Setting the Tab Order

The tab order is the order by which pressing the **Tab** key moves focus from control to control on the form. While the form is being built the tab order is determined by the order in which you place the controls on the form. If the controls are rearranged you may nee to manually reset the tab order. To set the tab order:



- View the desired form in the **Form** window
- Open the **View** menu
- Choose **Tab Order**
- Select the desired control from the list
- Click and drag it into the required position

Filling a Control

A list box or combo box control placed on the form is not functional until the data that will appear on the list is added.

This is done by writing code in the sub procedure associated with the **Initialize** event. This triggers when the form is loaded. The **AddItem** method is used to specify the text that appears in the list.

The code below shows items added to a combo box named cboCourses:

```
With cboCourses
    .AddItem "Excel"
    .AddItem "Word"
    .AddItem "PowerPoint"
End With
```

Adding Code to Controls

As seen, forms and their controls are capable of responding to various events. Adding code to forms and control events are accomplished the same way as adding code to events of other objects.

How to Launch a Form in Code

The **Show** method of the form object is used to launch a form within a procedure.

Creating a procedure to launch a form enables you to launch a form from a toolbar, or menu as well as from an event such as opening a workbook.

Below is the syntax used to launch a form:

FormName.Show

frmNewData.Show

Unit 9 Introduction to Data Access Objects

VBA is the programming language you use to interact with the Access Object Model. To interact with the data in a database, you will need to use a data access object model.

Access employs two such object models; Data Access Objects and ActiveX Data Objects.

DAO allows you to work with and manipulate the records in table and queries within an access database

Critical objects

Object	Use
Database	It is necessary to define the database that you are working with. You first define the database as a object Dim dbs as database And then assign the current database to it Set dbs = CurrentDB It is also possible to assign the variable to an external database
Recordsets	Recordsets are again first defined using a Dim statement Dim rst as DAO.Recordset You can then assign that recordset to a table or query Set rst = dbs.Openrecordset("Employees")

QueryDef	When you create a query in Access you are building a querydef A QueryDef is first defined Dim qdf as DAO.QueryDef It can then be assigned to an existing query, or to an SQL string Set Qdf = dbs.CreateQueryDef("Query Name")

We are primarily concerned with manipulating existing data. For this we are primarily concerned with using the Recordset object

Principle Methods of the RecordSet Object

Tasks	Description	
Opening	Dbs.OpenRecordset("Table1", dbOpenTable) This opens a table as a recordset	
Moving through a recordset	MoveFirst	Moves to first record
	MoveLast	Moves to last record
	MoveNext	Move to next record
	MovePrevious	Move to previous record
Editing a recordset	Edit	Prepares a record for editing
	Update	Commits the changes made to a record
Adding to a recordset	Append	Adds new data as a new record

Unit 10 Introduction to ADO (ActiveX Data Object)

Critical objects

Object	Use
Database Connection	It is necessary to define the connection that you are working with also if it is the current database You first define the Connection as a object Dim CN as ADODB.Connection And then assign the current database to it Set cn = CurrentProject.Connection It is also possible to assign the variable to an external database Set cn = New ADODB.Connection
Recordsets	Recordsets are again first defined using a Dim statement Dim rs as ADODB.Recordset You can then assign that recordset as a new recordset Set rs = New ADODB.recordset You can define the recordset by using a SQL string, a name of a table or a query from the current database strSql = "select * from tblImport" (SQL string all records from the table tblImport) and now you can open the recordset based on the SQL string rs.Open strSql, cn, adOpenDynamic, adLockOptimistic If you want to open a table as the recordset rs.Open "tblImport", cn, adOpenDynamic, adLockOptimistic If you want to open a recordset based on a query rs.Open "qryQuery", cn, adOpenDynamic, adLockOptimistic

We are primarily concerned with manipulating existing data. For this we are primarily concerned with using the Recordset object

Principle Methods of the RecordSet Object

Tasks	Description	
Moving through a recordset	MoveFirst	Moves to first record
	MoveLast	Moves to last record
	MoveNext	Move to next record
	MovePrevious	Move to previous record
Editing a recordset	Edit	Prepares a record for editing
	Update	Commits the changes made to a record
Adding to a recordset	Append	Adds new data as a new record

Some Examples how ADO code can look like:

The first example adds 20% to the column “Ord tot”

```
Public Sub CalcVat()

Dim cn As ADODB.Connection
Dim rs As ADODB.recordset
Dim strSql As String

Set cn = CurrentProject.Connection
Set rs = New ADODB.recordset

strSql = "select * from tblImport"
rs.Open strSql, cn, adOpenDynamic, adLockOptimistic


rs.MoveFirst

Do Until rs.EOF

rs![VAT] = rs![ord tot] * 0.2


rs.MoveNext

Loop

rs.Close
Set rs = Nothing



End Sub
```

The second example change the prices in a price list

```
Sub ChangeProductPrices()
Dim rs As ADODB.recordset
Dim sSQL As String
Dim sValue As String
Dim cn As ADODB.Connection
Set cn = CurrentProject.Connection

sSQL = "SELECT * FROM Items"
Set rs = New ADODB.recordset
rs.Open sSQL, cn, adOpenDynamic, adLockOptimistic

DoCmd.SetWarnings False
rs.MoveFirst
Do Until rs.EOF
If rs.Fields("product description") Like "shoe*" Then
rs.Fields("unit price") = rs.Fields("unit price") * 1.07
ElseIf rs.Fields("product description") Like "boots*" Then
rs.Fields("unit price") = rs.Fields("unit price") * 1.1
ElseIf rs.Fields("product description") Like "ball*" Then
rs.Fields("unit price") = rs.Fields("unit price") * 1.09
ElseIf rs.Fields("product description") Like "*fishing" Then
rs.Fields("unit price") = rs.Fields("unit price") * 1.09
End If
rs.MoveNext
Loop


DoCmd.SetWarnings True
rs.Close

End Sub
```

My Course Notes