

Access

VBA Programming

VBA code is written & stored within **modules**

There are two basic types of modules:

- those associated with **forms** or **reports**

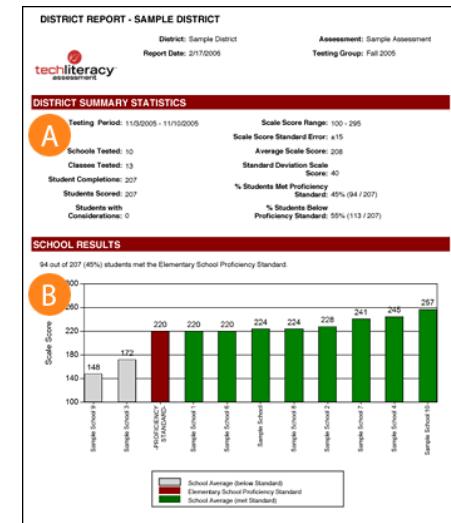
faxmemo2.frm

Attention:
Receiver's Name:
RECVR_1
Company, Address, City, ST, Zip
RECVR_2
RECVR_3
RECVR_4
Phone / FAX Numbers
RECVR_5
Email Address
RECVR_6

of Pages: (including this one.)
PAGE
Date: DATE_
Time: TIME_
File:
REF_1
REF_2
REF_3
CC:
CC

From:
Sender's Name:
SENDER_1
Company, Address, City, ST, Zip
SENDER_2
SENDER_3
SENDER_4
Phone / FAX Numbers
SENDER_5
Email Address
SENDER_6

MEMO / FAX Transmission



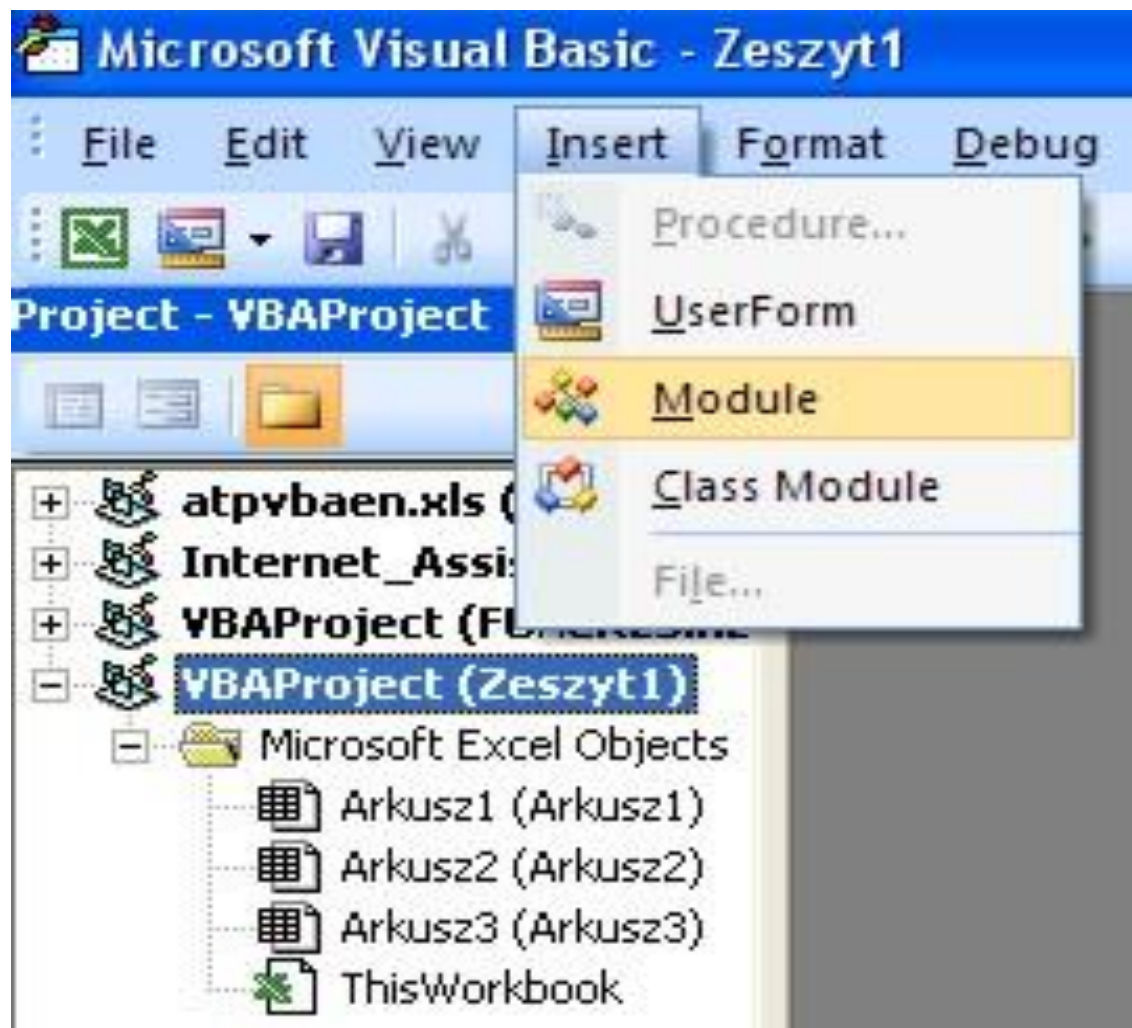
- & a **general module** for holding procedures that is applicable to the whole project.

There is one other type, called *class module*, which contains **code** associated with an object.

Before you can write VBA code, you need to be able to create the module. Since form or report modules are created through their respective forms & reports, we will focus on creating standard modules here.

There are a couple of **different ways of creating a standard module**. The easiest way is to use the Modules category of objects right in the DB Win, ...

Creating Standard Modules



Creating Procedures

Most VBA code is contained in blocks called procedures. Which are divided into two categories:

sub procedures (called subs) or
functions.

Most of the time, you will see the following two lines of code at the top of the module:

Option Compare Database

Option Explicit

These are called the **general declarations** of the module. Any code put into this section will affect all the procedures in the module.

Creating Procedures

The Option Compare line has 3 possible settings for sorting strings of characters within the code:

Option Compare Database This uses the same **sort order** as the DB itself & is the most common setting.

Option Compare Binary This **sorts a string** of characters based upon the **binary values of the characters**. Thus, uppercase letters would be recognized before lowercase.

Option Compare Text This makes the sort case sensitive with regard to the local language.

We will keep the Option Compare Database setting as default.

The **Option Explicit** line is an important one. As you shall see, that line can keep you from making a lot of coding mistakes by **forcing you to formally declare all variables before using them.**

Creating Procedures

Think of a **procedure as a miniprogram** to do one specialized job.

For instance, a procedure may be used to add two numbers. Every time those numbers need to be added, you can just call this prewritten procedure. Without the procedure, you might need to write redundant code every time you needed to do that particular job.

Creating Procedures

Within the procedure, you **declare variables**, have **loops** and **If** statements, and **call other procedures**.

Let's build a simple procedure that adds two numbers together:

A procedure is first declared with the following syntax:

```
Sub AddNumbers()
```

```
End Sub
```

(of course, the name is anything that you choose it to be):

Creating Procedures- practices

The first practice is to **name your procedures** with a name that reflects its job, for instance, AddNumbers. Any other programmers looking at it would easily discern what it does. In addition, even though VBA is not case sensitive, it will sometimes need to interact with programs that are.

A common **naming convention**, is to use lowercase letters, with the exception of midword capitalization, no spaces, and to begin with a letter rather than a number. In the case of **beginning with a letter** rather than a number, this is more than just a naming convention.

The second practice is to **indent the code** in the procedure. That way, it is easy to spot where the procedure begins and ends. The VBA Editor will preserve the indenting for subsequent lines.

The third practice is to carefully **comment your code**. You can easily do that by using the single quote before a line. If you do that, VBA will ignore that line as a comment.

Declaring Variables

Within a procedure, you will find two basic components: **variables** and **methods**. A ***variable*** is a piece of information stored somewhere in the computer's memory. It could be a number, a letter, or an entire sentence. The location where it is stored in memory is known by the **variable name**. As an example, let's say that you have a line of code like this:

```
number = 237
```

From that point on, every time the code refers to the variable name **number**, 237 will be put in its place.

In order for the variable to function properly, you should also declare what type of information is going to be kept by the variable.

Types of Variables

Variable Type	Description
Boolean	returns True or False , or it in terms of the numbers, 0 = False & -1 = True.
Byte	It can only hold a single value between 0 and 255 .
Currency	It holds a currency value with 4 decimal places: -922 337 203 685 477.5808 .. 922 337 203 685 477.5807
Date	This stores both dates and time . Interestingly, the years range from 100 to 9999 .
Double	Doubles are for very large numbers. The range runs from $-4.940656458411247 * 10^{-324}$ to $4.94065645841247 * 10^{324}$
Integer	Integer handles the range of numbers -32 768 .. 32 767
Long	Long handles the number range: -2 147 483 648 .. 2 147 483 657
Object	You can store an object as a variable for later use.
Single	This is the second decimal point type, the other being Double.
String	A String variable can hold up to 2 billion characters.

To declare a variable

you use the keyword Dim. As an example:

```
Dim number As Integer
```

This declares that the variable name `number` will hold only data of type Integer. You will notice that we have not assigned a value yet (and in fact the editor will not allow us to do so). We only declared the name and type. At some future point in the code, we would have a line like this:

```
number = 32
```

>>> VBA is not case sensitive. The variable names **number**, **Number** & **numBer** would all be recognized as the same name

Assigning values to a variable

First of all, a **variable of type String** *must* have its value enclosed in quotation marks " ".

For instance, this would be a proper assignment:

```
Dim lastName as String  
    lastName = "Smith"
```

The second thing you must be aware of is that **variables of type Date** must have their values enclosed in **# signs**. A proper assignment would be as follows:

```
Dim BirthDate as Date  
    BirthDate = #10/08/03#
```

The same rules apply to variables. These are not requirements, but conventions. The prefixes associated with variables are:

Table 6-2: Variable Prefixes	
Variable Type	Prefix
Boolean	bln
Byte	byt
Currency	cur
Date	dat
Double	dbl
Integer	int
Long	lng
Object	obj
Single	sng
String	str

Variant

A *Variant* allows VBA to make its own decision as to what type of data it is holding. It is the default variable type and is used automatically if you leave the “as type” clause off the variable’s declaration. It uses the prefix of var.

As an example, let’s say we declare a variable as follows:

Dim varMyData

Because we left out the “as type” parameter, this defaults to the type Variant and will be the same as if you typed:

Dim varMyData as Variant

Let’s say you assign it as follows:

varMyData = “This is a Variant”

VBA will convert varMyData into a String type. If, later on, you assign the following:

varMyData = 12

VBA will now convert varMyData into type Integer.

As we progress, you will see situations where a variant could end up being a type other than what was wanted or, even worse, **could result in an error**. Many programmers also argue that too many variants take up too much memory “overhead” and slow the code down. So before you make the decision to work with variants, you want to carefully weigh the pros and cons.

Constants

Many times you may want to declare a value that will not change. This type is called a *Constant* and is declared using the keyword **Const** in place of the normal declaration using Dim.

As an example:

```
Const cNumber1 As Integer
```

```
Const cDate      As Date = #03/02/04#
```

Notice that you preface the variable's name with con instead of the normal type. Also, when declaring a constant, you must assign it an initial value, or you will get a syntax error message when you leave the line.

Input and Output

How do you **get** the information into a variable? Or
read the information that is stored in them?

One of the simplest techniques for getting information from the user into a variable is to use a built-in function called **InputBox**. This will give you a simple dialog box with a prompt.

To code this in our small example, you would enter the shaded lines shown here:

```
Sub addNumbers()  
    'Declare the variables  
    Dim intNumber1 As Integer  
    Dim intNumber2 As Integer  
    Dim intSum As Integer  
    'Create InputBoxes to enter numbers  
    intNumber1 = InputBox("Enter the first number")  
    intNumber2 = InputBox("Enter the second number")  
End Sub
```

Input and Output

The screenshot displays the Microsoft Visual Basic for Applications environment. The main window shows the code editor for a module named 'addNumbers'. The code defines a subroutine that takes two integer inputs and calculates their sum. Annotations in green text explain parts of the code: 'Declare the variables' points to the variable declarations, and 'Create InputBoxes to enter numbers' points to the InputBox calls. Below the code editor, two Microsoft Access dialog boxes are shown, representing the input prompts. The first dialog box, titled 'Enter the first number', has the value '123' entered. The second dialog box, titled 'Enter the second number', has the value '333' entered. A blue arrow points from the 'Run' button in the VBA toolbar to the first input dialog box.

Microsoft Visual Basic for Applications - Database2 - [Module1 (Code)]

File Edit View Insert Debug Run Tools Add-Ins Window Help

Ln 9, Col 1

Project - Database2

Database2 (Database2)

Modules

Module1

(General) addNumbers

```
Sub addNumbers()  
    Dim intNumber1 As Integer, intNumber2 As Integer 'Declare the variables  
    Dim intSum As Integer  
  
    'Create InputBoxes to enter numbers  
    intNumber1 = InputBox("Enter the first number")  
    intNumber2 = InputBox("Enter the second number")  
  
End Sub
```

Microsoft Access

Enter the first number

OK Cancel

123

Microsoft Access

Enter the second number

OK Cancel

333

Input and Output

Sub addNumbers()

Dim iNum1 As Integer, iNum2 As Integer, iSum As Integer

'Create InputBoxes to enter numbers

iNum1 = InputBox("Enter the 1st number: ")

iNum2 = InputBox("Enter the 2nd number: ")

'Add numbers

iSum=iNum1 + iNum2

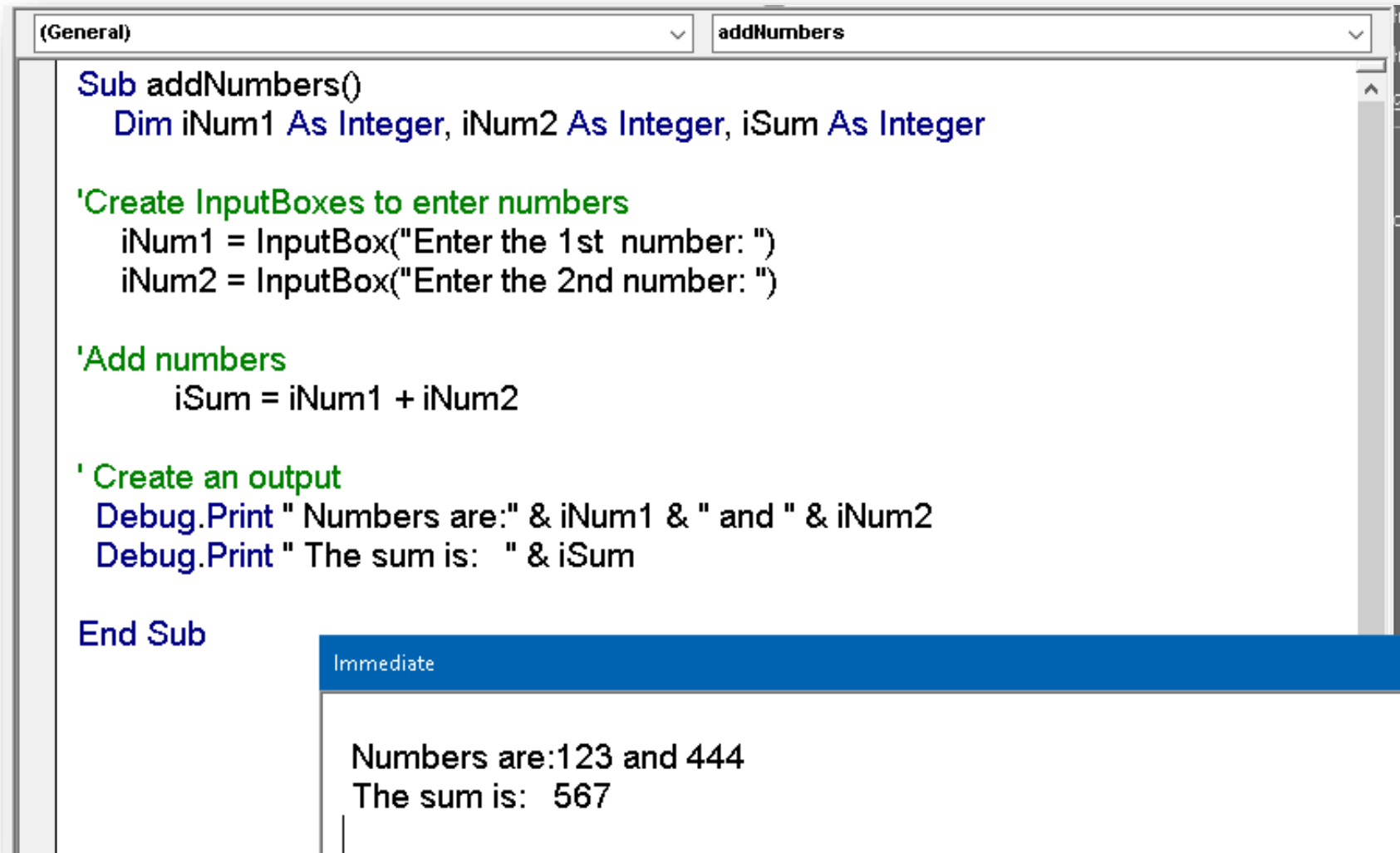
' Create an output

Debug.Print "Numbers are:" & iNum1 & " and " & iNum2

End Sub

MsgBox "Numbers are:" & iNum1 & " and " & iNum2

Input and Output



The screenshot shows a Visual Basic IDE window. The top bar has a dropdown menu set to '(General)' and another set to 'addNumbers'. The code editor contains the following VBA code:

```
Sub addNumbers()  
    Dim iNum1 As Integer, iNum2 As Integer, iSum As Integer  
  
    'Create InputBoxes to enter numbers  
    iNum1 = InputBox("Enter the 1st number: ")  
    iNum2 = InputBox("Enter the 2nd number: ")  
  
    'Add numbers  
    iSum = iNum1 + iNum2  
  
    ' Create an output  
    Debug.Print " Numbers are:" & iNum1 & " and " & iNum2  
    Debug.Print " The sum is:  " & iSum  
  
End Sub
```

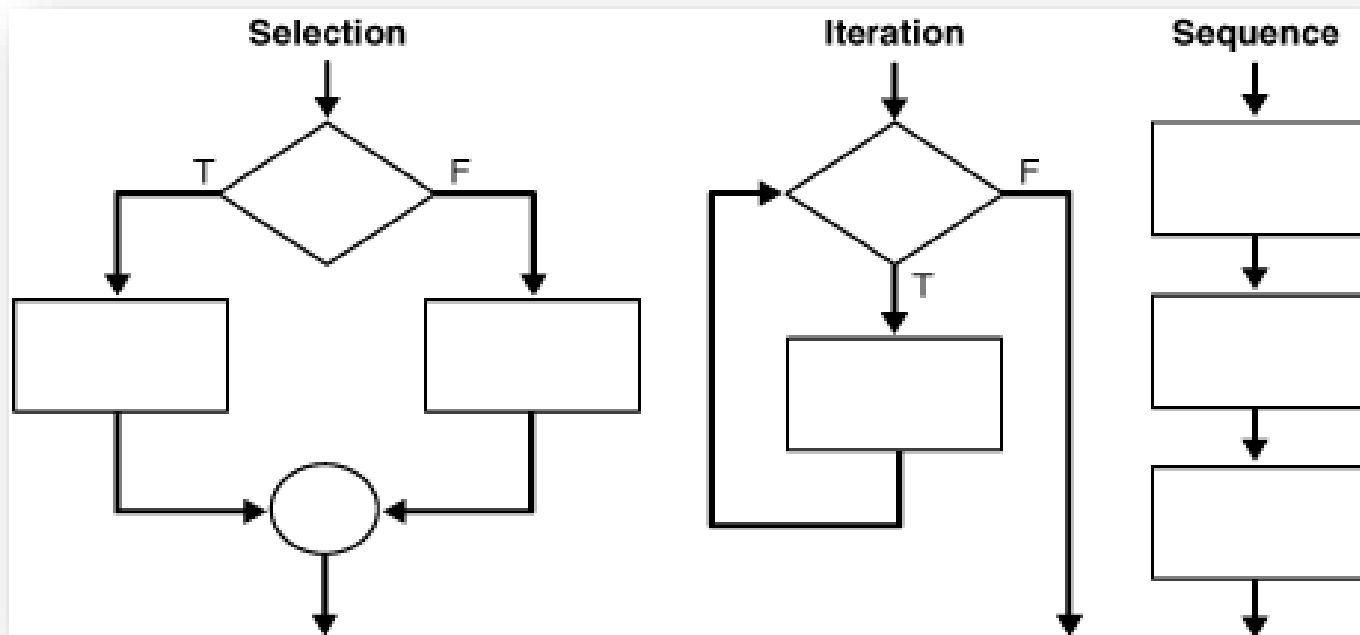
Below the code editor is the 'Immediate' window, which displays the output of the code:

```
Numbers are:123 and 444  
The sum is:  567
```

Control Structures

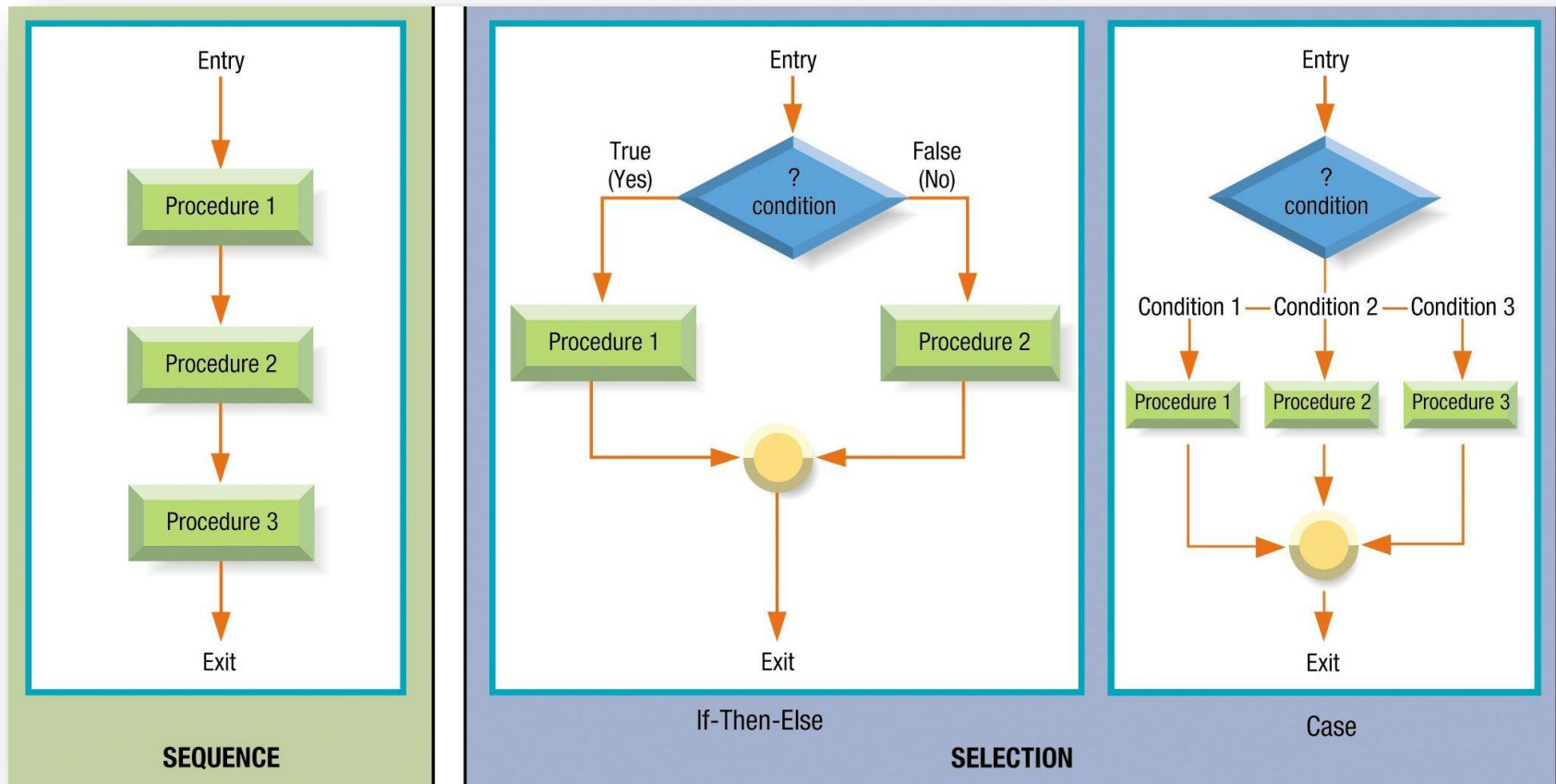
Computer code only runs in a sequence. It first executes a statement and, if there is no error, moves on to the next statement. We call this a ***sequence structure***. However, what happens if you want the next step in the sequence not to be the next line of code? You may need to transfer control of the program to a different block of code if a certain situation occurs. Or you may need a block of code to repeat until a situation occurs.

We call the mechanisms to accomplish this ***control structures***:
decision structures and **repetition structures**.



Control Structures - Decision structure

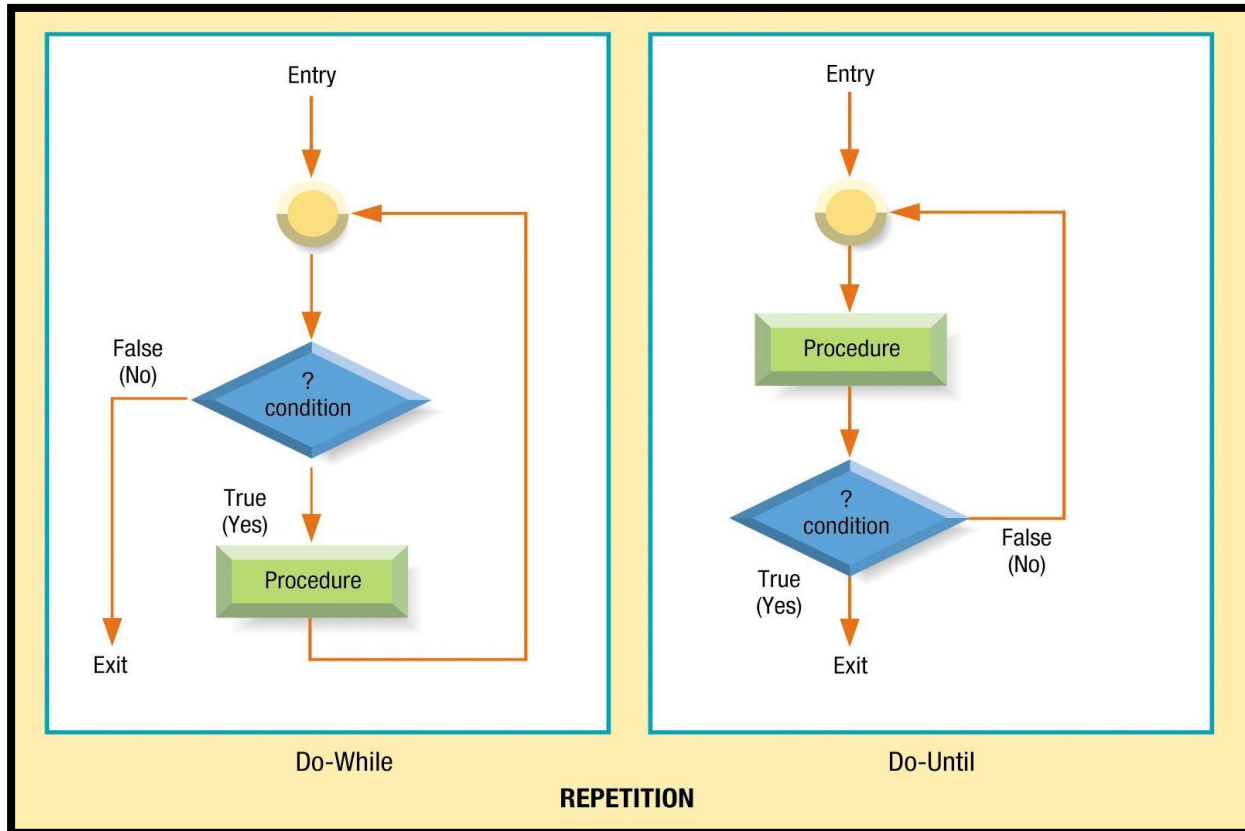
allows the program to make decisions. The most common of them is the **If...Then** structure. However, VBA provides other possibilities: **If...Then...Else**, **Elself**, **Select Case**, and **IIF**.



Control Structures - Repetition structures

goes through a block of code either a predefined number of times, or until something occurs to cause the program to break out of the loop.

VBA provides two repetition structures: **For...Next** and **Do...Loop**. Within those two structures, there are a number of variations.



Decision-Making Structures

Let's consider the pseudocode for a morning routine. You may have one point where you write the following:

If it is raining,

Then I will take an umbrella.

Else I will just go to my car.

The words in **boldface** are the VBA keywords necessary to make a decision.

Conditional expressions

You start a decision-making structure by doing a comparison of two entities:

Is entity A equal to entity B?

Is entity A greater than entity B?

Is entity B true?

These are called

conditional expressions

=	Is equal to
<>	Is not equal to
>	Is greater than
<	Is less than
>=	Is greater than or equal to
<=	Is less than or equal to

Simple Decision-making Structure In A Subroutine

```
Sub ifTest()
```

```
    Dim iNum As Integer , sMessage As String
```

```
    iNum = 12
```

```
    If iNum > 10 Then
```

```
        sMessage = "The number is " & iNum
```

```
    End If
```

```
    Debug.Print sMessage
```

```
End Sub
```

Simple Decision-making Structure in a Subroutine

```
Sub if_Test()
```

```
    Dim iNum As Integer , sMessage As String
```

```
    iNum = 8
```

```
    If iNum > 9 Then
```

```
        sMessage = "The number is greater than 9,,
```

```
    Else
```

```
        sMessage = "The number is less than 9"
```

```
    End If
```

```
    Debug.Print sMessage
```

```
End Sub
```

Simple Decision-making Structure In A Subroutine

```
Sub ifTest()
```

```
Dim intNum As Integer
```

```
intNum = 8
```

```
If intNum = 1 Then
```

```
    Debug.Print "This is the lowest number"
```

```
Elseif intNum = 15 Then
```

```
    Debug.Print "This is the highest number"
```

```
Else
```

```
    Debug.Print "The number is between 1 and 15"
```

```
End If
```

```
End Sub
```

```
Sub ifTest()  
    Dim intNum as Integer  
    intNum = 2  
    Select Case intNum  
    Case 1  
        Debug.Print "This is the lowest number"  
    Case 15  
        Debug.Print "This is the highest number"  
    Case Else  
        Debug.Print "The number is between 1 and 15,,  
    End Select  
End Sub
```

If you have multiple cases in which you want to carry out the same set of instructions, you can use the syntax:

Case 1, 2, 3...

Or you could use

Case 1 To 4

IIF

There is one other structure that we will take a quick look at: **IIF**. This is referred to as the **Immediate If**. This is handy if you want to assign the final value to a variable because its syntax is self-contained. The correct syntax is

IIF(*conditional test*, value for True, value for False)

So, working code might look something like this:

```
strMessage = IIF(intNum > 9, "Number is > 9", "Number < 9")
```

The performance of the IIF structure is somewhat slow and rarely used by programmers in a larger programming project.

Loops

We use loops when you need to have a block of code repeated a certain number of times or until an event of some sort happens.

The number of times the code repeats can be controlled by a counter. This is called *counter-controlled repetition*. The second type is called *sentinel-controlled repetition*. We will look at both types.

For...Next Loop

The For...Next loop is an example of a **counter-controlled repetition**. Using either actual numbers, or variables, you can set the following components for the loop:

Counter - the heart of the loop. It tracks the number of times the loop has repeated.

Start - the starting number for the counter. It is rare, to set this to anything else but 1.

End - number that marks the end of the loop, where you want the counter to stop.

Step You can specify a number for the counter to increment with each loop. This part is optional.

An example of the syntax for a For...Next loop:

```
Dim intCounter As Integer
```

```
For intCounter = 1 To 25
```

```
.....
```

```
Next
```

Do Loop

Do loop - it will continue until a specific condition occurs There are two variations of the Do loop—the **Do While** and the **Do Until**. We will look at each one.

Do While loop tests to see if a condition is true. If it is true, the statements in the loop execute. Let's look at the following subroutine:

```
Sub doTest()
```

```
    Dim intCounter As Integer, intTest As Integer
```

```
    intTest = 1
```

```
    intCounter = 1
```

```
    Do While intTest = 1
```

```
        Debug.Print "This is loop number " & intCounter
```

```
        If intCounter >= 5 Then
```

```
            intTest = 0
```

```
        End If
```

```
        intCounter = intCounter + 1
```

```
    Loop
```

```
End Sub
```



```
Sub ifTest2()  
    If intNum > 10 Then  
        MsgBox intNum & " is greater than 10"  
    Else  
        MsgBox intNum & " is less than 10"  
    End If  
End Sub
```

```
Sub ifTest()  
    Dim strMessage As String, intTest As Integer  
    intTest = 1  
    Do  
        intNum = InputBox("Enter a number between 1 and 15", _  
                           "Testing the If structure")  
        If intNum >= 1 And intNum <= 15 Then  
            intTest = 0  
            iftest2  
        Else  
            MsgBox "Sorry, the number must be between 1 and 15"  
        End If  
    Loop While intTest = 1  
End Sub
```

Sub doTest() ' 2nd ver

Dim intCounter As Integer, intTest As Integer

intTest = 1

intCounter = 1

Do Until intTest <> 1

Debug.Print "This is loop number " & intCounter

If intCounter >= 5 Then

intTest = 0

End If

intCounter = intCounter + 1

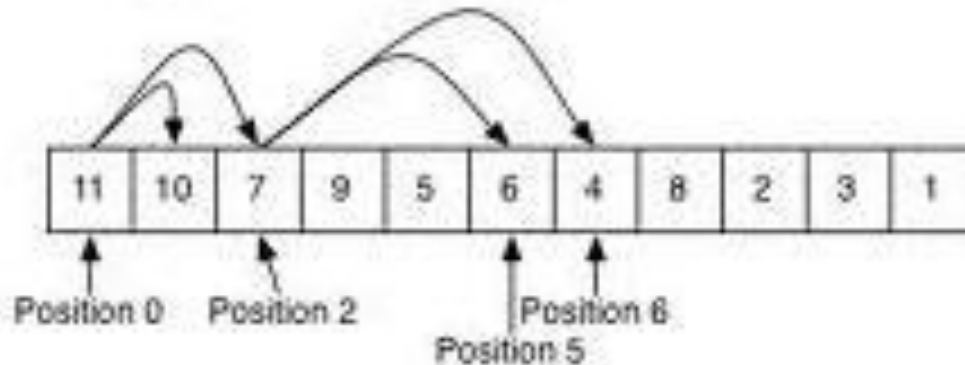
Loop

End Sub

Arrays

An *array* is a variable that contains multiple values. The number of values that the variable will hold must be decided and declared in advance.

You are also going to learn how to allocate memory properly so that your array does not take up too much room.



Components of an Array

Each value in the array is called an [element](#). Since an array variable has multiple elements in it, you need a way to pick them out or reference them individually. You can do that by using a number called an [index](#). **The first element of an array is index number 0.**

If you could look behind the scenes of an array of names, using variable name `strName`, it might look something like this:

```
strName    (0) "John Smith"  
           (1) "Jane Doe"  
           (2) "Rosemary Brown"  
           (3) "Anita LaScala"  
           (4) "Bob Gray"
```

Notice that even though the index numbers only go up to 4, this is a five-element array. Again, the **first element usually begins at 0.**

If you wanted to select Anita LaScala's name out of the array for printing, you would use:

```
Print strName(3)
```

VBA gives us two flavors of arrays

- **Static array** The number of elements in the array, called the length of the array, is decided in advance & remains **fixed**.
- **Dynamic array** The length of the array is **variable** & not decided in advance.

Static Arrays

A static array has a predetermined length and does not change. Declaring a static array is similar to declaring a variable:

Dim MyScores(10) As Integer

What we really did was declare an array of 11 elements, with the first element being 0, and the last element being index number 10 (sometimes called the *upper bound*). The *lower bound*, or lowest index number, of this array is 0.



In this procedure, you are going to create two **For...Next** loops. The first one will allow you to populate the array, and the second will print the contents of the array back to you:

Sub **arrayTest()**

```
Dim i As Integer
```

```
Dim MyScores(9) As Integer
```

```
For i = 0 To 9
```

```
    MyScores(i) = InputBox("Enter number:" & i, "Static Array Test")
```

```
Next
```

```
For i = 0 To 9
```

```
    Debug.Print "For array element" & i & "the number is" & MyScores(i)
```

```
Next
```

```
End Sub
```

Dynamic Arrays

Many programmers consider the concept of a dynamic array in VBA a slight programming fiction. Essentially, **it is still a static array, but you do not declare the size until the program is running.**

So the only issue is *when* the size is being declared. You start off by declaring an empty array:

```
Dim intMyScores() As Integer
```

Then you use a keyword, **ReDim**, to **redeclare** the size of the array while the program is running and it is known what the size will be.

Dynamic Arrays

Let's redo our previous example to **demonstrate a dynamic array**:

```
Sub ReDim_ArrayTest()
```

```
Dim i As Integer
```

```
Dim iMyScores() As Integer, iArraySize As Integer
```

```
iArraySize = InputBox("How many scores are you entering?", "Array Size")
```

```
ReDim iMyScores(1 to iArraySize)
```

```
For i = 1 To iArraySize
```

```
    iMyScores(i) = InputBox("Enter number " & i, "Static Array Test")
```

```
Next
```

```
For i = 1 To iArraySize
```

```
    Debug.Print "For array element" & i & "the number is" & iMyScores(i)
```

```
Next
```

```
End Sub
```

Notice that we first declare iMyScores as an empty array. Then, we use the **ReDim** keyword to redefine it as a static array, with the upper bound being controlled by iArraySize, which is entered by the user.

Erasing Arrays

You will sometimes have situations in which you want to keep the array declared, but erase the elements within it. You can easily do that with the keyword `Erase`, as shown here:

```
Erase intMyScores
```

This clears the contents but keeps the declaration.

Depending on the type of array, different things may happen. If it is a numeric array, the elements are set to 0. The elements of a string array are set to "" (an empty string). If it is a Boolean array, each element is set to False.

Documenting Your Code

It is a good idea to **document your code very well** so that you and others will be able to understand the purpose of each procedure and why it was written in a certain way. You can document your code in several ways. Here is an example of a general comment section”

```
*****
```

```
'* Procedure Name: TestProcedure
```

```
'* Purpose: The purpose of this procedure is to illustrate why  
'* documenting your code is so important.
```

```
'* Parameters: None.
```

```
'* Date Created: September 4, 2006 '* By: Denise M. Gosnell
```

```
'* Modification Log:
```

```
'* Date Explanation of Modification Modified By
```

```
'* -----
```

```
*****
```

Error Handling

Various types of errors can occur in your VBA code. How to debug them?

Default Error Messages

You have all used applications that did not have very good error handling and found yourself getting booted out of the application without warning or faced with unfriendly messages like the ones you saw earlier in this chapter. You do not want such problems to happen when **users interact with your application**.

Errors will always occur,

but if you design error handlers correctly, at least they can be handled reasonably well.

Handling Errors with an **On Error** Statement

On Error Statement

can be placed on your procedure to specify a section in the procedure to which the execution should jump when an error occurs. An example:

```
Sub Test()
```

```
    On Error GoTo HandleError
```

```
    'normal code for the procedure goes here
```

```
    Exit Sub
```

```
    HandleError:
```

```
    'code for handling the error goes here
```

```
    Exit Sub
```

```
End Sub
```

Visual Basic Editor

Opening the **VBA Editor** - **ALT + F11**

The **Project Explorer** (**CTRL+R**) window contains the list & the hierarchy of code objects for the project.

The **Properties** window (**F4**) contains a list of properties associated with an object, such as a form or report, and what their current settings are.

The **Object Browser** (**F2**) lets you examine all of the objects available to the project. Notice the word “available.” You may or may not use them. **Within each object, you will see the properties, methods, and events associated with that object.**

F5, F7, (^)F8, ^G, ^M, ^S, Alt+Enter

The Object Browser (F2)

OB lets you examine all of the objects available to the project.

You may or may not use them. Within each object, you will see the properties, methods, and events associated with that object.

You will also be able to see a description.



- **properties** (icon showing a hand holding a card)



- **methods** (icon for looks like a brick in motion)



- **classes**

Objects are entities in the computer's memory.

However, they have to come from somewhere.



The source of an object is a *class*, which serves as a kind of template from which an object is created.

On the left of the browser, you will see an enormous number of classes.

The actual classes and members that are defined in our current project are bolded.

later >> Create & Show the **Form_frmCustomer** class.

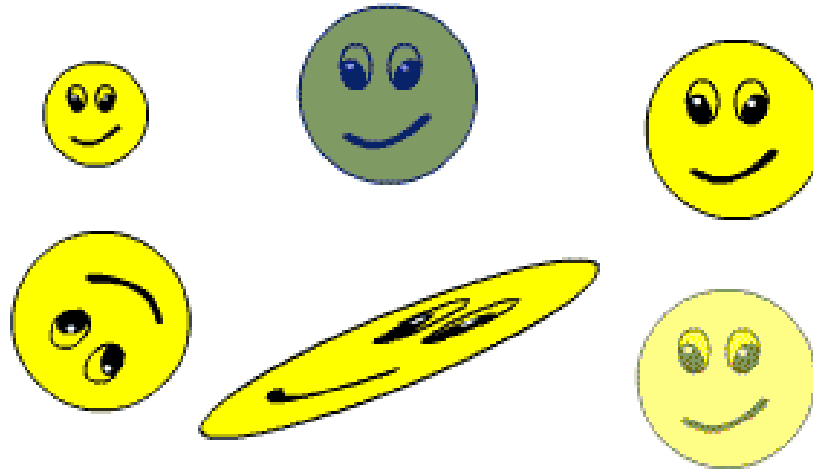
The right side of the browser shows the **members of the class**.

Classes have 3 main types of members: properties, methods (subs & functions), and events (which are a *special kind of sub*).

Properties are roughly the equivalent of a field. In true object-oriented programming, a field is sometimes referred to as an *attribute*:

Field \approx *attribute*

an object = **an instance** of a class



>> show the **AllowDeletions** attribute. Note the icon to the left of the attribute.