CHAPTER 3: PROGRAM STRUCTURE

Format of a visual program

1. **Object:** An object is a combination of code and data that can be treated as a unit. An object can be a piece of an application, like a control or a form. An entire application can also be an object. Objects are things that you can program with, things that make programming easier. They contain a set of methods and properties that allow you to make the object do certain things without actually having to look at the objects code.

   Objects let you declare variables and procedures once and then reuse them whenever needed.

   Each object in Visual Basic is defined by a class. A class describes the variables, properties, procedures, and events of an object. Objects are instances of classes; you can create as many objects you need once you have defined a class.

   To understand the relationship between an object and its class, think of cookie cutters and cookies. The cookie cutter is the class. It defines the characteristics of each cookie, for example size and shape. The class is used to create objects. The objects are the cookies.

   Two examples in Visual Basic might help illustrate the relationship between classes and objects.
   • The controls on the Toolbox in Visual Basic represent classes. When you drag a control from the Toolbox onto a form, you are creating an object — an instance of a class.
   • The form you work with at design time is a class. At run time, Visual Basic creates an instance of the form's class — that is, an object.

2. **Code:** A key part of developing applications using Visual Basic is ensuring that the **code** is carefully structured. This involves segmenting the code into *projects*, *modules* and *procedures* so that it is easy to understand and maintain.

   A complete Visual Basic application is typically contained in a single project. Within a project, code is placed in separate code files called *modules*, and within each *module*, the Visual Basic code is further separated into self-contained and re-usable *procedures*

   Visual Basic procedures provide a way to break up code into logical and re-usable sections that can be called from other sections of Visual Basic code. For example, you might have a section of Visual Basic code that calculates the interest due on a loan. It is also possible that you need to perform this calculation from a number of different places in your application code. Rather than duplicating the code to perform this task at each code location where it is needed, it is more efficient to place the calculation code in a procedure, and then *call* that procedure each time it is needed.

Visual Basic provides two types of procedures:

- **functions** - Functions are procedures which perform a task and return a value when completed.

- **subroutines** - Subroutines are procedures which perform a task but return no value when completed.

It is especially useful to be able to return values from *functions*. For example, the function may need to return the result of the task it performed (perhaps the result of a calculation). A function might also return a *True* or *False* value to indicate when the task was performed successfully. The Visual Basic code which called the function then acts based on the returned value.

In the case of both *subroutines* and *functions*, values (known as *parameters*) may optionally be passed into the procedure.

The syntax for a Visual Basic subroutine is as follows:

Scope ***Sub*** *subroutineName(parameters)*
End Sub

The *scope* value is either *Private* or *Public* depending on whether the Subroutine is to be accessible from Visual Basic code outside of the current module. *Sub* is the Visual Basic keyword which indicates this is a Subroutine rather than a Function. *subroutineName* is the name of the Subroutine and is used when this specific procedure is to be called. The *parameters* value allows the parameters accepted by this Subroutine to be declared

The *Public* keyword indicates the *scope*. This defines whether this subroutine is accessible from Visual Basic code residing in other modules. Setting the *scope* to *Private* would make the Subroutine inaccessible to Visual Basic code outside the current module.

The *Sub* keyword indicates that this is a Subroutine (as opposed to a Function) and as such, does not return a value on completion. Finally, the name of the Subroutine is provided. The parentheses are used to hold any parameters which may be passed through to the Subroutine when it is called. The *End Sub* code marks the end of the Subroutine. The Visual Basic code that constitutes the Subroutine is placed after the Subroutine declaration and the *End Sub*.

Data Types, Modules and Operators

Visual Basic uses building blocks such as Variables, Data Types, Procedures, Functions and Control Structures in its programming environment. This section concentrates on the programming fundamentals of Visual Basic with the blocks specified.

Modules

Code in Visual Basic is stored in the form of modules. The three kind of modules are Form Modules, Standard Modules and Class Modules. A simple application may contain a single Form, and the code resides in that Form module itself. As the application grows, additional Forms are added and there may be a common code to be executed in several Forms. To avoid the duplication of code, a separate module containing a procedure is created that implements the common code. This is a standard Module.

Class module (.CLS filename extension) are the foundation of the object oriented programming in Visual Basic. New objects can be created by writing code in class modules. Each module can contain:

Declarations : May include constant, type, variable and DLL procedure declarations.

Procedures : A sub function, or property procedure that contain pieces of code that can be executed as a unit.

These are the rules to follow when naming elements in VB - variables, constants, controls, procedures, and so on:

- A name must begin with a letter.
- May be as much as 255 characters long (but don't forget that somebody has to type the stuff!).
- Must not contain a space or an embedded period or type-declaration characters used to specify a data type; these are ! # % $ & @
- Must not be a reserved word (that is part of the code, like Option, for example)
- The dash, although legal, should be avoided because it may be confused with the minus sign. Instead of First-name use First_name or FirstName.

Data types

By default Visual Basic variables are of variant data types. The variant data type can store numeric, date/time or string data. When a variable is declared, a data type is supplied for it that determines the kind of data they can store. The fundamental data types in Visual Basic including variant are integer, long, single, double, string, currency, byte and boolean. Visual Basic supports a vast array of data types. Each data type has limits to the kind of information and the minimum and maximum values it can hold. In addition, some types can interchange with some other types. A list of Visual Basic's simple data types are given below.

1. Numeric

Byte	Store integer values in the range of 0 - 255
Integer	Store integer values in the range of (-32,768) - (+ 32,767)

Long	Store integer values in the range of (- 2,147,483,468) - (+ 2,147,483,468)
Single	Store floating point value in the range of (-3.4x10-38) - (+ 3.4x1038)
Double	Store large floating value which exceeding the single data type value
Currency	store monetary values. It supports 4 digits to the right of decimal point and 15 digits to the left

2. String

Use to store alphanumeric values. A variable length string can store approximately 4 billion characters

3. Date

Use to store date and time values. A variable declared as date type can store both date and time values and it can store date values 01/01/0100 up to 12/31/9999

4. Boolean

Boolean data types hold either a true or false value. These are not stored as numeric values and cannot be used as such. Values are internally stored as -1 (True) and 0 (False) and any non-zero value is considered as true.

5. Variant

Stores any type of data and is the default Visual Basic data type. In Visual Basic if we declare a variable without any data type by default the data type is assigned as default.

Operators

Arithmetical Operators

Operators	Description	Example	Result
+	Add	5+5	10
-	Substract	10-5	5
/	Divide	25/5	5
\	Integer Division	20\3	6
*	Multiply	5*4	20
^	Exponent (power of)	3^3	27
Mod	Remainder of division	20 Mod 6	2

&	String concatenation	"George"&" "&"Bush"	"George Bush"

Relational Operators

Operators	Description	Example	Result
>	Greater than	10>8	True
<	Less than	10<8	False
>=	Greater than or equal to	20>=10	True
<=	Less than or equal to	10<=20	True
<>	Not Equal to	5<>4	True
=	Equal to	5=7	False

Logical Operators

Operators	Description
OR	Operation will be true if either of the operands is true
AND	Operation will be true only if both the operands are true

Variable

a variable is an area of computer memory you use in your program. To use a variable, you must give it a name. There are rules you should, and usually must, follow when naming your variables. The name of a variable:

- Must begin with a letter
- Cannot have a period (remember that we use the period to set a property; in other words the period is an operator)
- Can have up to 255 characters. Please, just because it is allowed, don't use 255 characters.
- Must be unique inside of the procedure or the module it is used in (we will learn what a module is)

Once a variable has a name, you can use it as you see fit. For example, you can assign it a value and then use the variable in your program as if it represented that value.

Variable Declaration

In Visual Basic, it is a good practice to declare the variables before using them by assigning names and data types. They are normally declared in the general section of the codes' windows using the **Dim** statement. You can use any variable to hold any data , but different types of variables are designed to work efficiently with different data types .
The syantax is as follows:

Dim VariableName As DataType

If you want to declare more variables, you can declare them in separate lines or you may also combine more in one line , separating each variable with a comma, as follows:

Dim *VariableName1* As *DataType*1, *VariableName2* As *DataType*2,*VariableName3* As *DataType*3

Example
Dim password As String
Dim firstnum As Integer
Dim secondnum As Integer
Dim total As Integer
Dim doDate As Date
Dim password As String, yourName As String, firstnum As Integer

Unlike other programming languages, Visual Basic actually doesn't require you to specifically declare a variable before it's used. If a variable isn't declared, VB willautomatically declare the variable as a Variant. A variant is data type that can hold any type of data.

For string declaration, there are two possible types, one for the variable-length string and another for the fixed-length string. For the variable-length string, just use the same format as example 5.1 above. However, for the fixed-length string, you have to use the format as shown below:

Dim VariableName as String * n, where n defines the number of characters the string can hold.

Example

Dim yourName as String * 10

yourName can holds no more than 10 Characters.

Scope of Declaration

Other than using the Dim keyword to declare the data, you can also use other keywords to declare the data. Three other keywords are private ,static and public. The forms are as shown below:

Private *VariableName as* Datatype
Static *VariableName as* Datatype
Public *VariableName as* Datatype

The above keywords indicate the scope of declaration. Private declares a local variable, or a variable that is local to a procedure or module. However, Private is rarely used, we normally use Dim to declare a local variable. The Static keyword declares a variable that is being used multiple times, even after a procedure has been terminated. Most variables created inside a procedure are discarded by Visual Basic when the procedure is finished, static keyword preserve the value of a variable even after the procedure is terminated. Public is the keyword that declares a global variable, which means it can be used by all the procedures and modules of the whole program.

CHAPTER 4: PROGRAM WRITING

Define design time, run time, and break time.

- **Design time** refers to the period of development in which you carefully plan and design the user interface. You also write out the pseudocode and the actual code during design time.
- When you are testing and debugging your program, you are in the midst of **run time**.
- If you experience an error that stops your program, you experience what is called **break time**.

Steps to Develop a Program

The following steps are used in sequence for developing an efficient program:

- Specifying the problem statement
- Designing an algorithm
- Coding
- Debugging
- Testing and Validating
- Documentation and Maintenance.

Specifying the Problem:

The Problem which has to be implemented into a program must be thoroughly understood before the program is written. Problem must be analyzed to determine the input and output requirements of the program. A problem is created with these specifications.

Designing an Algorithm:

With the problem statement obtained in the previous step, various methods available for obtaining the required solution are analyzed and the best suitable method is designed into algorithm.

To improve clarity and understandability of the program flow charts are drawn using the algorithms.

Coding:

The actual program is written in the required programming language with the help of information depicted in flow charts and algorithms.

Debugging:

There is a possibility of occurrence of errors in programs. These errors must be removed to ensure proper working of programs. Hence error check is made. This process is known as “Debugging”.

Types of errors that may occur in the program are:

- Syntactic Errors: These errors occur due to the usage of wrong syntax for the statements.

Syntax means rules of writing the program.

Example: x=z*/b;

There is syntax error in this statement. The rules of binary operators state that there cannot be more than one operator between two operands.

- Runtime Errors: These Errors are determined at the execution time of the program.

Example: Divide by zero

Range out of bounds

Square root of a negative number

- Logical Errors: These Errors occur due to incorrect usage of the instruction in the program. These errors are neither displayed during compilation or execution nor cause any obstruction to the program execution. They only cause incorrect outputs. Logical Errors are determined by analyzing the outputs for different possible inputs that can be applied to the program. By this way the program is validated.

Testing and Validating:

Testing and Validation is performed to check whether the program is producing correct results or not for different values of input.

Documentation and Maintenance:

Documentation is the process of collecting, organizing and maintaining, in written the complete information of the program for future references. Maintenance is the process of upgrading the program according to the changing requirements.

For writing up the instructions as a program in the way that a computer can understand, we use programming languages.

Debug, Compile and Execution Terms

Debugging is the process of locating and fixing or bypassing bugs (errors) in computer program code. Programs sometimes have small errors, called "bugs," in them. These bugs can be minor, such as not recognizing user input, or more serious, such as a memory leak that crashes the program.

Compiling is the transformation from Source Code (human readable) into machine code (computer executable). A compiler is a program. A compiler takes the recipe (source code) for a new program (written in a high level language) and transforms this Code into a new language (Machine Language) that can be understood by the computer itself. This "machine language" is difficult to impossible for humans to read and understand (much less debug and maintain), thus the need for "high level languages" such as VB.

Compiler terminology

- **Compile** Colloquially, to convert a source code file into an executable, but strictly speaking, compilation is an intermediate step

- **Link** The act of taking compiled code and turning it into an executable

- **Build** A build refers to the process of creating the end executable (what is often colloquially referred to as compilation). Tools exist to help reduce the complexity of the build process--makefiles, for instance.

- **Compiler** Generally, compiler refers to both a compiler and a "linker"

- **Linker** The program that generates the executable by linking

- **IDE I**ntegrated **D**evelopment **E**nvironment, a combination of a text editor and a compiler, such that you can compile and run your programs directly within the IDE. IDEs usually have facilities to help you quickly jump to compiler errors.

Execute a program is to run the program in the computer, and, by implication, to start it to run. Process by which a computer or a virtual machine performs the instructions of a computer program

CHAPTER 5: CONTROL STRUCTURES

Introduction to Control Structures

Control structures allow you to control the flow of your program's execution. If left unchecked by control-flow statements, a program's logic will flow through statements from left to right, and top to bottom. While some very simple programs can be written with only this unidirectional flow, and while some flow can be controlled by using operators to regulate precedence of operations, most of the power and utility of any programming language comes from its ability to change statement order with structures and loops.

Control structures form the basic entities of a "**Visual programming language**". ***Control structures are used to alter the flow of execution of the program.*** Why do we need to alter the program flow? The reason is "***decision making***"! In life, we may be given with a set of option like doing "Electronics" or "Computer science". We do make a decision by analyzing certain conditions (like our personal interest, scope of job opportunities etc.). With the decision we make, we alter the flow of our life's direction. This is exactly what happens in a VB program. We use control structures to make decisions and alter the direction of program flow in one or the other path(s) available.

Types of Control Structures

There are **three types** of control structures available in VB

1) Sequence structure (straight line paths)

2) Selection structure (one or many branches)

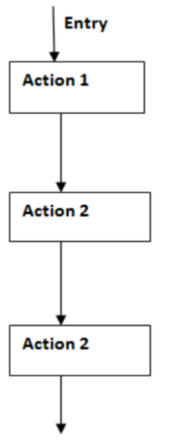
3) Loop structure (repetition of a set of activities)

All the 3 control structures and its flow of execution is represented in the flow charts given below.

Sequence structure

In a **sequence structure**, an action, or event, leads to the next ordered action in a predetermined order. The **sequence** can contain any number of actions, but no actions can be skipped in the **sequence**.

Implementation of sequence structure lead to a program execution from start to end in order of program statements



Sequence Structure

To implements Other "control structures" in a VB program, the language provides 'control statements'. So to implement a particular control structure in a programming language, we need to learn how to use the relevant control statements in that particular language.

The control statements are:-

- **Switch**
- **If**
- **If Else**
- **While**
- **Do While**
- **For**

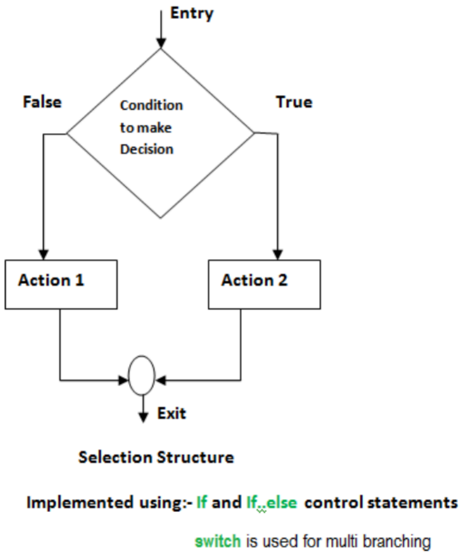
As shown in the flow charts:-

- Selection structures are implemented using **If** , **If Else** and **Switch** statements.
- Looping structures are implemented using **While**, **Do While** and **For** statements.

Selection/Decision/Branching structure

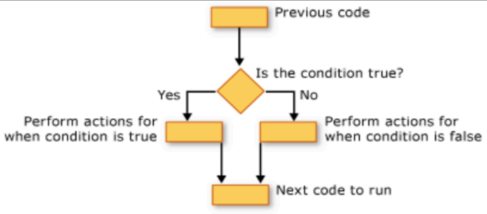
Selection structures are used to perform ‘**decision making**‘ and then branch the program flow based on the outcome of decision making. Selection structures are implemented in C/C++ with If, If Else and Switch statements. If and If Else statements are 2 way branching statements where as Switch is a multi branching statement.

Selection structures chart


Entry
False
Condition to make Decision
True
Action 1
Action 2
Exit
Selection Structure
Implemented using:- If and If..else control statements
switch is used for multi branching

Visual Basic lets you test conditions and perform different operations depending on the results of that test. You can test for a condition being true or false, for various values of an expression, or for various exceptions generated when you execute a series of statements.

The following illustration shows a decision structure that tests for a condition being true and takes different actions depending on whether it is true or false.


Previous code
Is the condition true?
Yes
No
Perform actions for when condition is true
Perform actions for when condition is false
Next code to run

Taking different actions when a condition is true and when it is false

Decision Structures Implementation

A branch is a point at which your program must make a choice. With these data structures, it is now possible to make programs that can have multiple outcomes. You may be familiar with the first type from Algebra, an If-Then statement, or *If P then Q*. And they work pretty much the same. Another type is the *Select Case*, this may prove to be easier at times.

Another name for this concept is *conditional clauses*. Conditional clauses are blocks of code that will only execute if a particular expression (the condition) is true.

If...Then Statement

If...Then statements are some of the most basic statements in all of programming. Every language has them, in some form or another. In Visual Basic, the syntax for an If...Then statement is as follows:

```
If (condition) Then
    (reaction)
End If
```

- **condition** - a set of test(s) that the program executes.
- **reaction** - the instructions that the program follows when the condition returns true. The condition returns true if it passes the test and returns false if it fails the test.

The condition can be anything from

```
If X = 1 Then
    MsgBox "X = 1"
End If
```

to

```
If InStr(1, sName, " ") > 0 Then
    sName = """" & sName & """"
End If
```

If there is only one reaction to the condition, the statement can also be expressed without the End If:

```
If Y + 3 = 7 Then MsgBox "Y + 3 DOES = 7"
```

There are also other parts to these statements to make them more complex. Two other terms that can be used are Else, and ElseIf.

Else will, if the condition is false, do whatever comes between the Else statement and the End If statement.

ElseIf will, if the condition **directly proceeding it** is false, check for another condition and go from there.

If..Then..Else Statement

The If..Then..Else statement is the simplest of the conditional statements. They are also called branches, as when the program arrives at an "If" statement during its execution, control will "branch" off into one of two or more "directions". An If-Else statement is generally in the following form:

```
  If condition Then
      statement
  Else
      other statement
  End If
```

If the original condition is met, then all the code within the first statement is executed. The optional Else section specifies an alternative statement that will be executed if the condition is false. The If-Else statement can be extended to the following form:

```
  If condition Then
      statement
  ElseIf condition Then
      other statement
  ElseIf condition Then
      other statement
      ...
  Else
      another statement
  End If
```

Only one statement in the entire block will be executed. This statement will be the first one with a condition which evaluates to be true. The concept of an If-Else-If structure is easier to understand with the aid of an example:

```
  Dim Temperature As Double
  ...
  If Temperature >= 40.0 Then
      Debug.Print "It's extremely hot"
  ElseIf 30.0 <= Temperature And Temperature<=39.0 Then
      Debug.Print "It's hot"
```

```
ElseIf 20.0 <= Temperature And Temperature <= 29.0 Then
    Debug.Print "It's warm"
ElseIf 10.0 <= Temperature And temperature <= 19.0 Then
    Debug.Print "It's cool"
ElseIf 0.0 <= Temperature And Temperature <= 9.0 Then
    Debug.Print "It's cold"
Else
    Debug.Print "It's freezing"
End If
```

Optimizing hints

When this program executes, the computer will check all conditions in order until one of them matches its concept of truth. As soon as this occurs, the program will execute the statement immediately following the condition and continue on, without checking any other condition for truth. For this reason, when you are trying to optimize a program, it is a good idea to sort your If..Then..Else conditions in order of descending likelihood. This will ensure that in the most common scenarios, the computer has to do less work, as it will most likely only have to check one or two *branches* before it finds the statement which it should execute. However, when writing programs for the first time, try not to think about this too much lest you find yourself undertaking premature optimization.

In Visual Basic Classic conditional statements with more than one conditional do not use short-circuit evaluation. In order to mimic C/C++'s short-circuit evaluation, use ElseIf as described in the example above. In fact for complicated expressions explicit *If..Then..ElseIf* statements are clearer and easier to read than the equivalent short circuit expression.

Select Case

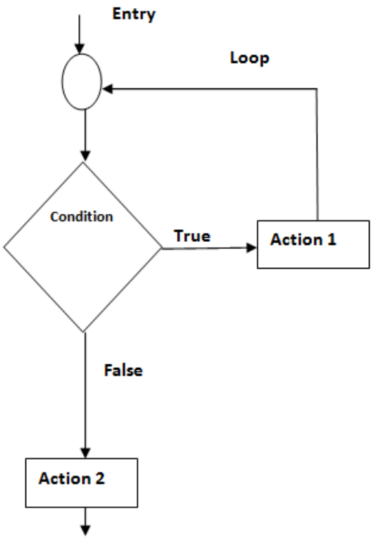
Often it is necessary to compare one specific variable against several constant expressions. For this kind of conditional expression the Select Case is used. The above example is such a case and could also be written like this:

```
Select Case Temperature
    Case Is >= 40#
        Debug.Print "It's extremely hot"
    Case 30# To 39#
        Debug.Print "It's hot"
    Case 20# To 29#
        Debug.Print "It's warm"
    Case 10# To 19#
        Debug.Print "It's cool"
    Case 0# To 9#
        Debug.Print "It's cold"
    Case Else
        Debug.Print "It's freezing"
End Select
```

Loop Structure

A loop structure is used to execute a certain set of actions for a predefined number of times or until a particular condition is satisfied. There are 3 control statements available in C/C++ to implement loop structures. **While, Do while and For statements.**

Loop structures chart



Loop Structure

Implemented using:- While , Do While and For control statements

Loop Structure Implementation

Loops are control structures used to repeat a given section of code a certain number of times or until a particular condition is met.

Visual Basic has three main types of loops: *for..next* loops, *do* loops and *while* loops.

Note: 'Debug' may be a reserved word in Visual Basic, and this may cause the code samples shown here to fail for some versions of Visual Basic.

For..Next Loops

The syntax of a *For..Next* loop has three components: a *counter*, a *range*, and a *step*. A basic *for..next* loop appears as follows:

```
For X = 1 To 100 Step 2
 Debug.Print X
Next X
```

In this example, X is the counter, "1 to 100" is the range, and "2" is the step.

The variable reference in the *Next* part of the statement is optional and it is common practice to leave this out. There is no ambiguity in doing this if code is correctly indented.

When a *For..Next* loop is initialized, the counter is set to the first number in the range; in this case, X is set to 1. The program then executes any code between the *for* and *next* statements normally. Upon reaching the *next* statement, the program returns to the *for* statement and increases the value of the counter by the step. In this instance, *X* will be increased to *3* on the second iteration, *5* on the third, etc.

To change the amount by which the *counter* variable increases on each *iteration*, simply change the value of the *step*. For example, if you use a *step 3*, *X* will increase from *1* to *4*, then to *7, 10, 13*, and so on. When the step is not explicitly stated, *1* is used by default. (Note that the *step* can be a negative value. For instance, `for X = 100 to 1 step -1` would decrease the value of X from 100 to 99 to 98, etc.)

When X reaches the end of the range in the range (100 in the example above), the loop will cease to execute, and the program will continue to the code beyond the *next* statement.

It is possible to edit the value of the counter variable within a *for..next* loop, although this is generally considered bad programming practice:

```
For X = 1 To 100 Step 1
 Debug.Print X
 X = 7
Next
```

While you may on rare occasions find good reasons to edit the counter in this manner, the example above illustrates one potential pitfall:

Because X is set to 7 at the end of every iteration, this code creates an infinite loop. To avoid this and other unexpected behavior, use extreme caution when editing the counter variable!

It is not required by the compiler that you specify the name of the loop variable in the *Next* statement but it will be checked by the compiler if you do, so it is a small help in writing correct programs.

For loop on list

Another very common situation is the need for a loop which *enumerates* every element of a list. The following sample code shows you how to do this:

```
Dim v As Variant

For Each v In list
 Debug.Print v
Next
```

The list is commonly a Collection or Array, but can be any other object that implements an *enumerator*. Note that the iterating variable has to be either a **Variant**, **Object** or class that matches the type of elements in the list.

Do Loops

Do loops are a bit more flexible than For loops, but should generally only be used when necessary. Do loops come in the following formats:

- Do while
- Do until
- Loop while
- Loop until

While loops (both do while and loop while) will continue to execute as long as a certain conditional is true. An Until loop will loop as long as a certain condition is false, on the other hand. The only difference between putting either While or Until in the Do section or the Loop section, is that Do checks when the loop starts, and Loop checks when the loop ends. An example of a basic loop is as follows:

```
Do
 Debug.Print "hello"
 x = x + 1
Loop Until x = 10
```

This loop will print hello several times, depending on the initial value of x. As you may have noticed, Do loops have no built in counters. However, they may be made manually as shown above. In this case, I chose x as my counter variable, and every time the loop execute, x increase itself by one. When X reaches 10, the loop will cease to execute. The advantage of Do loops is that you may exit at any time whenever any certain conditional is met. You may have it loop as long as a certain variable is false, or true, or as long as a variable remains in a certain range.

Endless loop: *Do..Loop*

The endless loop is a loop which never ends and the statements inside are repeated forever. Never is meant as a relative term here - if the computer is switched off then even endless loops will end very abruptly.

```
Do
 Do_Something
Loop
```

In Visual Basic you cannot label the loop but you can of course place a label just before it, inside it or just after it if you wish.

Loop with condition at the beginning: Do While..Loop

This loop has a condition at the beginning. The statements are repeated as long as the condition is met. If the condition is not met at the very beginning then the statements inside the loop are never executed.

```
Do While X <= 5
 X = X + 5
Loop
```

Loop with condition at the end: Do..Loop Until

This loop has a condition at the end and the statements are repeated until the condition is met. Since the check is at the end the statements are at least executed once.

```
Do
 X = 5+2
Loop Until X > 5
```

Loop with condition in the middle:Do..Exit Do..Loop

Sometimes you need to first make a calculation and exit the loop when a certain criterion is met. However when the criterion is not met there is something else to be done. Hence you need a loop where the exit condition is in the middle.

```
Do
 X = Calculate_Something
 If X > 10 then
  Exit Do
 End If
 Do_Something (X)
Loop
```

In Visual Basic you can also have more than one exit statement. You cannot exit named outer loops using *Exit Do* because Visual Basic does not provide named loops; you can of course use Goto instead to jump to a label that follows the outer loop.

While Loops

While loops are similar to Do loops except that the tested condition always appears at the top of the loop. If on the first entry into the loop block the condition is false, the contents of the loop are never executed. The condition is retested before every loop iteration.

An example of a While loop is as follows:

```
    price = 2

    While price < 64
        Debug.Print "Price = " & price
        price = price ^ 2
    Wend

    Debug.Print "Price = " & price & ":  Too much for the market to bear!"
```

The While loop will run until the condition tests false - or until an "Exit While" statement is encountered.

Nested Loops

A nested loop is any type of loop inside an already existing loop. They can involve any type of loop. For this, we will use For loops. It is important to remember that the inner loop will execute its normal amount multiplied by how many times the outer loop runs. For example:

```
For i = 1 To 10
 For j = 1 To 2
  Debug.Print "hi"
 Next
Next
```

This will print the word 'hi' twenty times. Upon the first pass of the *i* loop, it will run the *j* loop twice. Then on the second pass of the *i* loop, it will run the *j* loop another two times, and so on.

CHAPTER 6: ERROR HANDLING & DEBUGGING

Introduction to Error Handling

Error handling refers to the anticipation, detection, and resolution of **programming**, application, and communications errors. Specialized **programs**, called **error** handlers, are available for some applications.

Types of Program Errors

A program with anything more than a trivial amount of code will almost inevitably contain errors (or "bugs") during its development. The types of errors commonly encountered can be categorised as follows:

- *Syntax errors* - these occur when a command is mis-typed or an expected argument is omitted. Visual Basic detects such errors as they occur, and will not allow the program to be run until they are corrected.
- *Run-time errors* - these are usually errors beyond the control of the programmer, such as the input of invalid data by the user or a missing data file. Error-trapping routines can be implemented to deal with such occurrences.
- *Logic errors* - this type of error is the most difficult to find, as it will not be automatically detected by visual basic. The program will often run, but may produce unexpected results. The Visual Basic debugger can be used to track down errors in the program's logic.

The incidence of errors can be reduced if care is taken with the design of the program. Time spent at the design stage may save considerably more time tracking down program errors later. Use of appropriate comments throughout the code will help you to remember what each section of code is doing, should you need to revisit the code to track down an error. The use of a consistent and meaningful naming convention for variables, controls and procedures will also aid clarity during the fault-finding process.

Although the following sections describe a number of very useful debugging tools and techniques, always look for the obvious answer before setting up some elaborate debugging procedure. If you do not know exactly where the problem is, one approach is to set a breakpoint (see below) somewhere in the middle of your program. If the program breaks before the error manifests itself, you can be fairly certain its in the second half of the program code. Otherwise, its in the first half. You can continue to narrow down the search in this way until the error is found.

Error handling techniques

This chapter describes various error handling techniques. First the technique is described, then its use is shown with an example function and a call to that function. We use the √ function which should report an error condition when called with a negative parameter.

Return code

```
function √ (X : in Float) : Float
```

```
begin
   if (X < 0) :
      return -1
   else
      calculate root from x
   fi
end
C := √ (A^2 + B^2)

if C < 0  then
   error handling
else
   normal processing
fi
```

Our example make use of the fact that all valid return values for √ are positive and therefore -1 can be used as an error indicator. However this technique won't work when all possible return values are valid and no return value is available as error indicator.

Error (success) indicator parameter

An error condition is returned via additional *out* parameter. Traditionally the indicator is either a boolean with "true = success" or an enumeration with the first element being "Ok" and other elements indicating various error conditions.

```
function √ (
   X       : in Float;
   Success : out Boolean ) : Float
begin
   if (X < 0) :
      Success := False
   else
      calculate root from x
      Success := True
   fi
end
C := √ (A^2 + B^2, Success)

if not Success then
   error handling
else
   normal processing
fi
```

This technique does not look very nice in mathematical calculations.

Global variable

An error condition is stored inside a global variable. This variable is then read directly or indirectly via a function.

```
function √ (X : in Float) : Float
begin
  if (X < 0) :
     Float_Error := true
  else
```

```
     calculate root from x
  fi
end
Float_Error := false
C := √ (A² + B²)

if Float_Error then
   error handling
else
   normal processing
fi
```

As you can see from the source the problematic part of this technique is choosing the place at which the flag is reset. You could either have the callee or the caller do that.

Also this technique is not suitable for multithreading.

Exceptions

The programming language supports some form of error handling. This ranges from the classic `ON ERROR GOTO ...` from early Basic dialects to the `try ... catch` exceptions handling from modern object oriented languages.

The idea is always the same: you register some part of your program as error handler to be called whenever an error happens. Modern designs allow you to define more than one handler to handle different types of errors separately.

Once an error occurs the execution jumps to the error handler and continues there.

```
function √ (X : in Float) : Float
begin
   if (X < 0) :
      raise Float_Error
   else
      calculate root from x
   fi
end
try:
   C := √ (A² + B²)
   normal processing
when Float_Error:
   error handling
yrt
```

The great strength of exceptions handling is that it can block several operations within one exception handler. This eases up the burden of error handling since not every function or procedure call needs to be checked independently for successful execution.

Error handling statements

Some useful error handling statements are there in Visual Basic 6 which help you ignore, bypass or handle errors in your program. Three such statements are helpful. They are as follows:

- **On Error Resume Next statement:** If any error occurs, it is ignored, and the control goes to the next statement.
- **On Error Goto label:** If any error occurs, the control jumps to a label.
- **On Error Goto 0:** This statement cancels the effect of 'On Error Resume Next' and 'On Error Goto label' statements.

On Error Resume Next

If Visual Basic encounters an error, it ignores the error. Then the control goes to the next statement. More precisely, Visual Basic causes a jump to the next statement. And Visual Basic executes the statements ignoring the statement where the error is found. Consider this example.

Example:

```
On Error Resume Next
a = 6 / 0

Print "hello"
```

On Error Goto label

If Visual Basic encounters an error in a statement, the On Error Goto label statement tells Visual Basic to jump to the named label. Thus the control jumps to the named label. Then the code following the named label is executed. Consider the following example.

Example:

```
On Error GoTo A
A = 6 / 0

A:
Print "hello"
Print "Welcome"
```

On Error Goto 0

On Error Goto 0 statement tells Visual Basic to cancel any effect of 'On Error Resume Next' and 'On Error Goto label' statements. So this statement cancels the effect of error handling in your program.

The Err function

The Err function can help you handle the error using the error code. For example, you can display a warning message to the end-user. The following example clarifies this.

Example:

```
On Error Resume Next
b = 88 / 0

If Err = 11 Then
   MsgBox "Error: Division by zero!"
End If
```

Example:

```
On Error GoTo Label5
b = 88 / 0

Label5:
If Err = 11 Then
   MsgBox "Error: Division by zero!"
End If
```

Design by Contract

In Design by Contract (DbC) functions must be called with the correct parameters. This is the caller's part of the contract. If the types of actual arguments match the types of formal arguments, and if the actual arguments have values that make the function's preconditions True, then the subprogram gets a chance to fulfill its postcondition. Otherwise an error condition occurs. Now you might wonder how that is going to work. Let's look at the example first:

```
function √ (X : in Float) : Float
    pre-condition (X >= 0)
    post-condition (return >= 0)
begin
   calculate root from x
end
C := √ (A² + B²)
```

As you see the function demands a precondition of `X >= 0` - that is the function can only be called when $X \geq 0$. In return the function promises as postcondition that the return value is also ≥ 0.

In a full DbC approach, the postcondition will state a relation that fully describes the value that results when running the function, something like `result ≥ 0 and X = result * result`. This postcondition is √'s part of the contract. The use of assertions, annotations, or a language's type system for expressing the precondition `X >= 0` exhibits two important aspects of Design by Contract:

1. There can be ways for the compiler, or analysis tool, to help check the contracts. (Here for example, this is the case when `X ≥ 0` follows from X's type, and √'s argument when called is of the same type, hence also ≥ 0.)
2. The precondition can be mechanically checked **before** the function is called.

The 1st aspect adds to safety: No programmer is perfect. Each part of the contract that needs to be checked by the programmers themselves has a high probability for mistakes.

The 2nd aspect is important for optimization — when the contract can be checked at compile time,

no runtime check is needed. You might not have noticed but if you think about it: is never negative, *provided* the exponentiation operator and the addition operator work in the usual way.

We have made 5 nice error handling examples for a piece of code which never fails. And this is the great opportunity for controlling some runtime aspects of DbC: You can now safely turn checks off, and the code optimizer can omit the actual range checks.

DbC languages distinguish themselves on how they act in the face of a contract breach:

1. True DbC programming languages combine DbC with exception handling — raising an exception when a contract breach is detected at runtime, and providing the means to restart the failing routine or block in a known good state.
2. Static analysis tools check all contracts at analysis time and demand that the code written in such a way that no contract can ever be breached at runtime.

Debugging & Debugging tools

Errors in a program's logic often do not prevent it from running, but may cause unexpected results. Visual Basic provides *debugging* tools to help the programmer track down such errors. The programmer's interface to the debugging tools is via three debug windows:

1. the *Immediate Window*
2. the *Locals Window*
3. the *Watch Window*

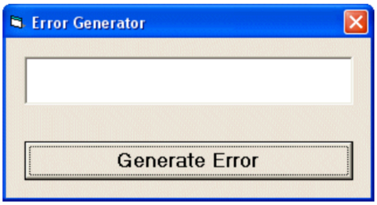
These windows can be accessed from the **View** menu or using the **Debug Toolbar** (accessed using the **Toolbars** option in the **View** menu). Debugging is carried out when the application is in **break**

mode. Break mode is entered by setting breakpoints, pressing **Ctrl+Break**, or when the program encounters an untrapped error or a **Stop** statement. In this mode, the debug windows and other tools can be used to:

1. determine the value of variables
2. set breakpoints
3. set watch variables and expressions
4. manually control the application
5. determine which procedures have been called
6. change the values of variables and properties

The debugging tools available in Visual Basic include *breakpoints*, *watch points*, *calls*, *step into*, *step over*, and *step out*.

Example 2
This example is simply a form with a single command button that will be used to demonstrate the use of the debugging tools. The code attached to the button's **Click** event is a simple loop that repeatedly calls a function.



Enter the following code in the form's code window:

```
Private Sub cmdClickMe_Click()
    Dim X As Integer, Y As Integer
    X = 0
    Do
        Y = Fcn(X)
        X = X + 1
    Loop While X <= 20
End Sub

Function Fcn(X As Integer) As Integer
    Fcn = CInt(0.1 * X ^ 2)
End Function
```

This program passes X, which has an initial value of 0, to the function **Fcn**, which then computes a value for Y. Each iteration of the loop increases X by 1 until the value of X exceeds 20. Although the program does not actually do a lot, it is useful for demonstrating the debugging facilities in Visual Basic.

Using Debug.Print
The simplest debugging method is to use a **Debug.Print** statement to print the value of a procedure's variables to the *immediate* window while an application is running. To demonstrate the use of this technique, amend the code for the **cmdClickMe_Click()** procedure as shown below, and run the application.

```
Private Sub cmdClickMe_Click()
    Dim X As Integer, Y As Integer
    X = 0
    Do
        Y = Fcn(X)
        Debug.Print X; Y
        X = X + 1
    Loop While X <= 20
End Sub
```

An examination of the immediate window will reveal that value of X and Y has been printed for each iteration of the loop. This information can be used to ensure that the values of these variables are correct. Once you have seen the **Debug.Print** statement in action, remove it from the code.

Using a Breakpoint
The above example allowed the procedure to run to completion before displaying the value of variables in the immediate window. Sometimes it is useful to be able to stop a procedure during execution, examine the value of the procedure variables, and then resume execution. This can be done with *breakpoints*. A breakpoint is a line in the code that causes a break in program execution.

To set a breakpoint, position the cursor in the line of code where you want the program to break, and either press the **F9 key**, or click the **Breakpoint** button on the **Debug** toolbar or select **Toggle Breakpoint** from the **Debug** menu. The selected line will be highlighted. When the program runs, execution will halt on lines with breakpoints to allow you to check the value of variables and expressions in the immediate window. Program execution can be resumed by pressing the **F5** key, or clicking on the **Run** button on the toolbar. All breakpoints can be cleared by selecting **Clear All Breakpoints** from the **Debug** menu. Individual breakpoints can be toggled using **Toggle Breakpoint**.

To demonstrate the use of breakpoints, set a breakpoint on the **X = X + 1** line in the program, and run it. When the program stops, display the immediate window and type the following line of code:

```
Print X;Y
```

The values of these variables will appear in the debug window. Now restart the application, and print the new variable values.

The *locals* window shows the value of the variables used by the *current* procedure. As execution moves from one procedure to another, the contents of the locals window will change accordingly. Repeat the breakpoint example, and this time note that the values of X and Y also appear in the locals window.

Using Watch Expressions
The **Add Watch** option in the **Debug** menu lets you set up *watch expressions* for your program. The watch expressions can be the values of variables, or logical expressions. The values of watch expressions are displayed in the *watch* window. Watch expressions can be edited using the **Edit Watch** option in the **Debug** menu.

To demonstrate the use of a watch expression, set a breakpoint on the **X = X + 1** line in the program, set a watch expression for the variable X, and run the program. You will see that X appears in the watch window, and will have a new value each time the program is restarted.

Now, clear the breakpoint, add a watch on the expression **X = Y**, and set the **Watch Type** to **Break When Value Is True**. Now run the application, and note that it goes into break mode and displays the watch window whenever X = Y.

Using Call Stack
Selecting the **Call Stack** button from the toolbar (or from the **View** menu) will display all currently active procedures. Call stack can be useful when dealing with situations involving nested procedures, and can give you an indication of where you are in the program.

Do demonstrate the use of **Call Stack**, set a breakpoint on the **Fcn = Cint()** line and run the application, which will break at this line. Press the **Call Stack** button. It will tell you that you are currently in the **Fcn** procedure, which was called from the **cmdClickMe_Click()** procedure.

Using Step Into
When the program is at a breakpoint you can continue program execution one line at a time by pressing **F8** or by selecting **Step Into** from the **Debug** menu. This will allow you to watch how variables change (in the locals window), or how the form changes as you step through the program. You can also execute several lines at a time by clicking on a line below the current line and pressing **Ctrl+F8**, or by selecting **Run To Cursor** from the **Debug** menu.

To demonstrate this technique, set a breakpoint on the **Do** line in the program code and run the application. When the program breaks, use the **F8** key to single step through the program. You can also try using the **Run To Cursor** option described above.

Using Step Over and Step Out
When stepping through a program, you probably won't want to step through procedures that you already know function properly. The **Step Over** facility executes all the steps in a procedure together, rather than one step at a time. To step over a procedure in this manner, select **Step Over** from the **Debug** menu, or press **Shift+F8**.

To demonstrate the use of **Step Over**, single step through the program once and note what happens. Then single step through the program again until you are on the line that calls the **Fcn** function, and press **Shift+F8**. Note that the program did not single step through the function as it did the first time round.

If you are currently stepping through a procedure and wish to execute the remainder of the procedure immediately, select **Step Out** from the **Debug** menu, or press **Ctrl+Shift+F8**.

CHAPTER 7: SUB-PROGRAMS

Introduction to Sub-Program and Functions/Procedures

The natural structure of a Visual Basic program is as independent sections of code delimited by "**Private Sub** *ObjectName_Event*" and "**End Sub**" statements. Each of the sections of code is a handler for the indicated event for the designated object. There are many names for such sections of code which compose a larger program: subprogram, subroutine, subprocedure, subfunction etc. Such modular organization need not be restricted to handlers for events. Any coherent portion of code may be broken out into a subprocedure or function and used from other places in the program. An event handler is an example of a subprocedure. A function differs from a subprocedure in being able to return a value.

A subprocedure is of the form

```
    Private Sub Procedurename()
        Statement
        …
    End Sub
```

or

```
    Private Sub Procedurename( argument AS type, …)
        Statement
        …
    End Sub
```

A function is of the form

```
    Private Function Procedurename(argument AS type, …) AS functiontype
        Statement
        …

        Procedurename = …

    End Function
```

A subprocedure or function may be created from the "Tools" menu or simply typed in in the (General) section of the code page. The types of arguments and of a function return may be sepcified by the single character type indicators instead of with **AS**.

Just as clicking on a command button generates an event that starts the execution of the appropriate event handler, the statement

```
    Call Procedurename
```

If there are no arguments, or

```
    Call Procedurename( arguments )
```

If there are arguments starts the execution of the subprocedure. Once the subprocedure has run, execution continues at the statement after the call. The number of arguments in the call should match the number of arguments in the declaration, and while Basic will do some conversions automatically, it is best if the types match as well.

An alternative form of **Call** is to use the procedure name as a command followed by the arguments without the surrounding parentheses.

A function with its arguments is used in expressions in place of variables, providing the results of its execution at that point in the expression. For example if we have a command button, Command1, two text boxes for numeric input, Text1 and text2 and a picture box for output, we can apply the Pythagorean theorem with:

```
Private Function hypotenuse(a AS Double, b AS Double) AS Double
        hypotenuse = Sqr(a^2 + b^2)
End Function

Private Sub Command1_Click()

        Picture1.Cls
        Picture1.Print hypotenuse(Val#(Text1.Text),Val#(Text2.Text))
End Sub
```

There are several useful built-in functions, some of which are given in the following table:

Abs(arg)	Absolute value of arg
Asc(str$)	ASCII numeric code for the first character of str$
Atn(arg)	Arctangent of arg
Chr$(arg)	One character string for which arg is the ASCII numeric code
C*type*(arg)	Converts arg to the indicated type, where *type* is one of Bool, Byte, Cur, Date, Dbl, Int, Lng, Sng, Str or Var
Cos(arg)	Cosine of arg in radians
Exp(arg)	e to the power arg
Fix(arg)	Integer part of arg (by truncation)
Format (arg,str$)	String derived from arg using format string str$
Hex$(arg)	String representing arg as hexadecimal
InStr(str$,probe$)	Position of probe$ in str$ or 0
InStr(offset, str$,probe$)	As with Instr, but starting at offset
Int(arg)	Largest integer <= arg
Left$(str$,n)	Leftmost n characters of str$
Len(str$)	Length of str$
Log(arg)	Natural logarithm of arg
Mid$(str$,offset, n)	Substring of str$ of length n starting at offset. May be usede as the left hand side of an assignment
Oct$(arg)	String representing arg as octal

Right$(str$,n)	Rightmost n characters of str$
Rnd	Random number from 0 to 1
Sgn(arg)	-1 if arg is negative, 0 is arg is zero, 1 if arg is positive
Sin(arg)	Sine of arg in radians
Space$(n)	String os n spaces
Sqr(arg)	Square root of arg
Str$(arg)	String representing the number in arg
Tan(arg)	Tangent of arg in radians
Val(str$)	Number represented by str$

Types of subprograms

Private

Private Sub sets the scope so that subs in outside modules cannot call that particular subroutine. This means that a sub in Module 1 could not use the Call method to initiate a Private Sub in Module 2. (Note: If you start at the **Application** level, you can use **Run** to override this rule and access a Private Sub)

Private [*insert variable name*] means that the variable cannot be accessed or used by subroutines in other modules. In order to be used, these variables must be declared outside of a subroutine (usually at the very top of your module). You can use this type of variable when you have one subroutine generating a value and you want to pass that value on to another subroutine in the same module.

Dim [*insert variable name*] is used to state the scope inside of a subroutine (you cannot use Private in its place). Dim can be used either inside a subroutine or outside a subroutine (using it outside a subroutine would be the same as using Private).

Public

Public Sub means that your subroutine can be called or triggered by other subs in different modules. Public is the default scope for all subs so you do not need to add it before the word "sub". However, it does provide further clarity to others who may be reading your code. As a personal preference I do not type Public Sub unless I am creating an intricate program that has a bunch of subroutines with varying scopes (ie I have a mix of Public & Private subs).

Public [*insert variable name*] means that the variable can be accessed or used by subroutines in outside modules. These variables must be declared outside of a subroutine (usually at the very top of your module). You can use this type of variable when you have one subroutine generating a value and you want to pass that value on to another subroutine stored in a separate module.

Private and Public Examples

Let's look at an example of how Public and Private scopes interact with each other. For this example let's presume that we insert two modules into a new workbook and place the below code in their respective module.

In Module 1

```
Dim x As Integer
Public y As Integer

Sub Start_Process()

  x = 15
  y = 12

  Call Print_Values

End Sub

'=========================================================

Private Sub Display_Message()

  MsgBox "We ran all three subroutines!"

End Sub
```

In Module 2

```
Sub Print_Values()

Debug.Print x
Debug.Print y

Call Display_Message

End Sub
```

Before we run any of the subprograms, note that there are two variables x and y that are dimensioned outside of a subroutine. This means that their values can carry over into other subprograms. The variable x has a private scope so only subroutines in the same module can access it's value. The variable y has a public scope, meaning that subroutines inside and outside it's module can access it's value.

Let's start by running *Start_Process*. All this subprograms does is give x and y a value and then initiates the *Print_Values macro* to start running. We can Call *Print_Values* even thought it's not in the same module because it is a Public Sub.

Now let's hop on over to *Print_Values* . In this subprograms we are going to debug print the values of x and y to the immediate window (ctrl + g). Notice that when you try to print variable x it outputs nothing. This is because x does not exist in Module 2. Therefore, a new variable x was created in Module 2 and since we did not give this new x variable a value it's output was nothing.

Notice that when we print variable y's value the number 12 is shown in the Immediate Window. This is because Module 2 subroutines have access to the public variables declared in Module 1.

Now the last line in *Print_Values* is going to give us an error. This is because we are trying to initiate *Display_Message* from Module 1. Since *Display_Message* was declared as a private sub, *Print_Values* does not have the ability to initiate it. There are a few things we can do to fix this:

1. We could remove the word "Private" from *Display_Message*
2. We could replace "Private" with "Public" in *Display_Message*
3. We can use the Application level and instead of using **Call** we could write **Application.Run "Display_Message "** (this method serves as an override in case we wanted to keep *Display_Message* private in the eyes of other outside module subroutines)

Scope of variables

A **Variable's Scope** : The scope of a variable is the section of the application that can see and manipulate the variable. If a variable is declared within a procedure, only the code in the specific procedure has access to that variable. When the variable's scope is limited to a procedure it's called **local variable**.

e.g.

```
Private Sub Command1_Click()

Dim i as Integer

Dim Sum as Integer

For i=0 to 100 Step 2

Sum = Sum +i

Next
```

```
MsgBox “ The Sum is “& Sum

End Sub
```

A variable whose value is available to all procedures within the same Form or Module are called Form-wide or Module-wide (Global **variable**) and can be accessed from within all procedures in a component. In some situations the entire application must access a certain variable. Such variable must be declared as Public.

Lifetime of a Variable : It is the period for which they retain their value. Variables declared as Public exist for the lifetime of the application. Local variables, declared within procedures with the Dim or Private statement, live as long as the procedure.

You can force a local variable to preserve its value between procedure calls with the Static keyword. The advantage of using static variables is that they help you minimize the number of total variables in the application.

Variables declared in a Form outside any procedure take effect when the Form is loaded and cease to exist when the Form is unloaded. If the Form is loaded again, its variables are initialized, as if it’s being loaded for the first time.

CHAPTER 8: DATA STRUCTURES

Description of data structures

VB6 works with simple data types, like Integer and String variables. Although these data types are useful in their own rights, more complex programs call for working with data structures; **that is, groups of data elements that are organized in a single unit,** for working with, maintaining, and manipulating lists of complex data.
Examples of common data structures:

- Arrays
- Enumerations
- Constants
- Structures

Introduction to Arrays

By definition, an array is a variable with a single name that represents many different items. When we work with a single item, we only need to use one variable. However, if we have a list of items which are of similar type to deal with, we need to declare an array of variables instead of using a variable for each item

For example, if we need to enter one hundred names, it is difficult to declare 100 different names. Besides, if we want to process those data that involves decision making, we might have to use hundreds of if...then statements, this is a waste of time and efforts.So, instead of declaring one hundred different variables, we need to declare only one array. We differentiate each item in the array by using subscript, the index value of each item, for example, name(1), name(2), name(3) ....etc. , makes declaring variables more streamline.

Types/Dimension of an Array data structure

An array can be one-dimensional or multidimensional. A one-dimensional array is like a list of items or a table that consists of one row of items or one column of items.

A two-dimensional array is a table of items that make up of rows and columns. The format for a one-dimensional array is ArrayName(x), the format for a two dimensional array is ArrayName(x,y) and a three-dimensional array is ArrayName(x,y,z) . Normally it is sufficient to use a one-dimensional and two-dimensional array, you only need to use higher dimensional arrays if you need to deal with more complex problems. Let me illustrate the arrays with tables

Table 16.1. One dimensional Array

Student Name	Name(1)	Name(2)	Name(3)	Name(4)

Table 16.2 Two Dimensional Array

Name(1,1)	Name(1,2)	Name(1,3)	Name(1,4)

Name(2,1)	Name(2,2)	Name(2,3)	Name(2,4)
Name(3,1)	Name(3,2)	Name(3,3)	Name(3,4)

Declaring Array

We can use Public or Dim statement to declare an array just as the way we declare a single variable. The Public statement declares an array that can be used throughout an application while the Dim statement declares an array that could be used only in a local procedure.

Declaring one dimensional Array

The general syntax to declare a one dimensional array is as follow:

Dim arrayName(subscript) as dataType

where subs indicates the last subscript in the array.

When you declare an array, you need to be aware of the number of elements created by the Dim keyword. In the Dim arrayName(subscript) statement, *subscript* actually is a constant that defines the maximum number of elements allowed. More importantly, subs start with 0 instead of 1. Therefore, the Dim arrangeName(10) statement creates 11 elements numbered 0 to 11. There are two ways to overcome this problem, the first way is by uisng the keyword Option Base 1, as shown in Example 16.1.

Example 16.1

```
Option Base 1
Dim CusName(10) as String
```

will declare an array that consists of 10 elements if the statement Option Base 1 appear in the declaration area, starting from CusName(1) to CusName(10). Otherwise, there will be 11 elements in the array starting from CusName(0) through to CusName(10)

CusName(1)	CusName(2)	CusName(3)	CusName(4)	CusName(5)
CusName(6)	CusName(7)	CusName(8)	CusName(9)	CusName(10)

The second way is to specify the lower bound and the upper bound of the subscript using To keyword. The syntax is

Dim arrayName(lowerbound To upperbound) As dataType

Example

Dim Count(100 to 500) as Integer

declares an array that consists of the first element starting from Count(100) and ends at Count(500)

Example

```
Dim studentName(1 to 10) As String
Dim num As Integer
Private Sub addName()
For num = 1 To 10
studentName(num) = InputBox("Enter the student name","Enter Name", "", 1500,
4500)
If studentName(num)<>"" Then
Form1.Print studentName(num)
Else
End
End If
Next
End Sub
```

**The program accepts data entry through an input box and displays the entries in the form itself.

Example 16.4

```
Dim studentName(1 to 10) As String
Dim num As Integer
Private Sub addName( )
For num = 1 To 10
studentName(num) = InputBox("Enter the student name")
List1.AddItem studentName(num)
Next
End Sub
Private Sub Start_Click()
addName
End Sub
```

```
**The program accepts data entries through an InputBox and displays the

items in a list box.
```

Declaring two dimensional Array

The general syntax to declare a two dimensional array is as follow:

Dim ArrayName(Sub1,Sub2) as dataType

Example 16.5

If you wish to compute a summary of students involve in games according to different year in a high school, you need to declare a two dimensional array. In this example, let's say we have 4 games, football, basketball, tennis and hockey and the classes are from year 7 to year 12. We can create an array as follows:

Dim StuGames(1 to 4,7 to 12) As Integer

Will create an array of four rows and six columns, as shown in the following table:

Year	**7**	**8**	**9**	**10**	**11**	**12**
Football	StuGames (1,7)	StuGames (1,8)	StuGames (1,9)	StuGames(1,10)	StuGames(1,11)	StuGames(1,12)
Basket ball	StuGames (2,7)	StuGames (2,8)	StuGames (2,9)	StuGames(2,10)	StuGames(2,11)	StuGames(2,12)
Tennis	StuGames (3,7)	StuGames (3,8)	StuGames (3,9)	StuGames(3,10)	StuGames(3,11)	StuGames(3,12)
Hockey	StuGames (4,7)	StuGames (4,8)	StuGames (4,9)	StuGames(4,10)	StuGames(4,11)	StuGames(4,12)

Example 16.6
In this example, we want to summarize the first half-yearly sales volume for four products. Therefore, we declare a two dimension array as follows:

Dim saleVol(1 To 4, 1 To 6) As Integer

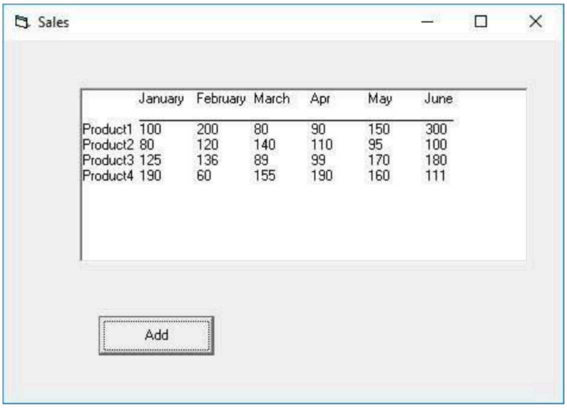
Besides that, we want to display the output in a table form. Therefore, we use a list box. We named the list box listVolume. AddItem is a listbox method to populate the listbox.

The code

```
Private Sub cmdAdd_Click()
Dim prod, mth As Integer ' prod is product and mth is month
Dim saleVol(1 To 4, 1 To 6) As Integer
Const j = 1
listVolume.AddItem vbTab & "January" & vbTab & "February" & vbTab
& "March" _
& vbTab & "Apr" & vbTab & "May" & vbTab & "June"
listVolume.AddItem vbTab &
"____________________________________________"
For prod = 1 To 4
For mth = 1 To 6
saleVol(prod, mth) = InputBox("Enter the sale volume for" & " " &
"product" & " " & prod & " " & "month" & " " & mth)

Next mth
Next prod
```

```
For i = 1 To 4

listVolume.AddItem "Product" & "" & i & vbTab & saleVol(i, j) &
vbTab & saleVol(i, j + 1) & vbTab & saleVol(i, j + 2) _
& vbTab & saleVol(i, j + 3) & vbTab & saleVol(i, j + 4) & vbTab &
saleVol(i, j + 5)

Next i

End Sub

*Note: the keyword vbTab is to create a space
```

The output is shown in the figure below:



```
         January  February  March  Apr   May   June
Product1 100      200       80     90    150   300
Product2 80       120       140    110   95    100
Product3 125      136       89     99    170   180
Product4 190      60        155    190   160   111
```

Figure 16.1

Dynamic Array

So far we have learned how to define the number of elements in an array during design time. This type of array is known as static array. However, the problem is sometimes we might not know how many data items we need to store during run time. In this case, we need to use dynamic array where the number of elements will be decided during run time. In VB6, the dynamic array can be resized when the program is executing. The first step in declaring a dynamic array is by using the Dim statement without specifying the dimenson list, as follows:

```
Dim myArray()
```

Then at run time we can specify the actual array size using the ReDim statement,as follows:

```
ReDim myArray(1 to n) when n is decided during run time
```

You can also declare a two dimensional array using ReDim statement, as follows:

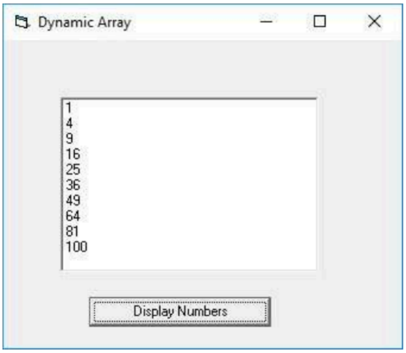
```
ReDim myArray(1 to n, 1 to m) when m and n are known during run time
```

Example 16.7
In this example, we want to display the elements of an array in a list box. The size of the array will only be known during run time. It demonstrates the creation of a dynamic array using the ReDim keyword.

The Code

```
Private Sub cmd_display_Click()

     Dim myArray() As Integer
     Dim i, n As Integer
     n = InputBox("Enter the upper bound of array")
     List1.Clear

     For i = 1 To n
     ReDim myArray(i)
     myArray(i) = i ^ 2
     List1.AddItem myArray(i)

     Next

    End Sub
```

The output



```
1
4
9
16
25
36
49
64
81
100
```

Figure 16.2

Another problem related to ReDim is each time you ReDim an array, the values currently stored in the array are lost. In order to preserve the stored values, we can use the keywords ReDim Preserve, as follows

```
            ReDim Preserve MyArray(n)
```

where n is the new upper bound of the array

Searching and Sorting Arrays

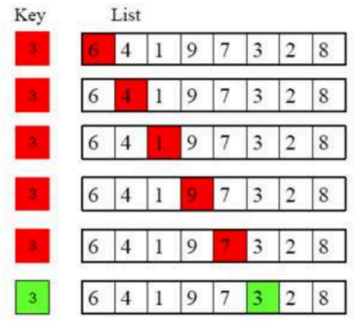
Searching Arrays

One of the most common tasks associated with arrays is searching through them to find some desired element.

Linear Searches

Suppose we wish to write a method that takes in a one-dimensional array and some value x, and returns either the position (i.e., "index") of x in the array, or -1 if x does not appear in the array.

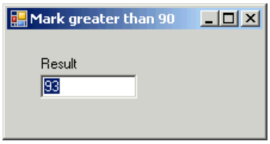
One option would be to use a **linear search**. In a linear search, we compare x (which we call the "key") with each element in the array list, starting at one end and progressing to the other. Graphically, we can imagine the following comparisons being made:



Supposing that our array is an array of integers, we might use the following method to accomplish this linear search:

Linear Search Example

Example in VB : An array of 10 exam marks is to be searched to find the one mark that was greater than 90...The mark is then displayed in a TextBox called txtResult



First the array has to be declared and values assigned to each element of the array....

Remember the maximum subscript is 9, which means there are 10 elements in the array.

```
Dim Mark(9) As Integer

Mark(0) = 65
Mark(1) = 48
Mark(2) = 78
Mark(3) = 60
Mark(4) = 39
Mark(5) = 71
Mark(6) = 93
Mark(7) = 55
Mark(8) = 62
Mark(9) = 51
```

There are two ways of carrying out a linear search...

Method 1 : involves using an integer variable (n) that counts the element being checked. The value of this counter needs to be incremented after each check...

```
Dim n As Integer

n = 0

Do

   If Mark(n) > 90 Then
      txtResult.Text = Mark(n)
   End If

   n = n + 1

Loop Until Mark(n - 1) > 90
```

Method 2 : Use a For...Next Loop...

```
Dim n As Integer

For n = 0 To 9

   If Mark(n) > 90 Then
      txtResult.Text = Mark(n)
   End If

Next
```

n is the Control Variable for the loop. It must be declared as an integer...and basically counts from 0 to 9 for each time the loop is executed.

Binary Searches

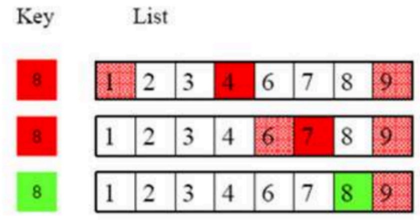
Linear Searches work well, but when used on arrays with a large number of elements, they potentially have to check *every* element to find a match. This can take a while, especially if we are doing multiple such searches.

There is a way to speed things up substantially, however -- provided that the elements in the array are ordered. (Let us assume for the sake of the discussion below that they are in ascending order).

A **binary search** works on an ordered list, and first compares the key with the element in the middle of the array. (*In the case of an even number of elements in our list, we will use the element that ends the first half of the list as our "middle element"*).

- If the key is less than the middle element, we only need to search the first half of the array, so we continue searching on this smaller list.
- If the key is greater than the middle element, we only need to search the second half of the array, so we continue searching on this smaller list.
- If the key equals the middle element, we have a match -- end the search

The diagram below illustrates how a key value of 8 is found in an ordered list in just 3 steps:



Note how we keep track of the sublist we are actually searching by keeping track of its left-most and right-most elements.

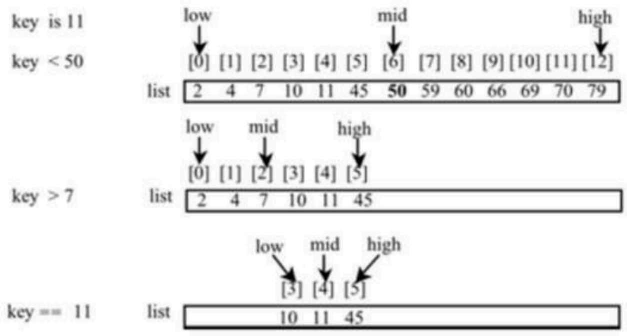
1. Initially, we are searching the entire list, so the left-most element is 1, while the right-most element is 9. There is an even number of elements in our list, so as mentioned above, we use the element that ends the first half of the list as our "middle element" (i.e., the "4").
2. Since 4 is less than the key value 8, we only need to search the second half of the list from 1 to 9. As such, we update the left-most element of the list searched to be the "6", while we leave the right-most element of the list searched unchanged (it's still the 9). The "middle element" of this sublist, which again has an even number of elements, is again taken as the element that ends the first half of this sublist from 6 to 9, which is 7.
3. Since 7 is still less than the key value 8, we again only need to search the second half of the sublist from 6 to 9. As such, we update the left-most element of the sublist searched to be the 8, while we leave the right-most element of the sublist searched unchanged (it's still the 9). The "middle

element" of this sublist, which again has an even number of elements, is again taken to be the element that ends the first half of this sublist from 8 to 9, which is 8. (*Note, as the sublist only has two elements, following the aforementioned rule for finding "middles", gives us a middle for this sublist identical to the "left-most" element of this sublist.*)

4. Since 8 equals the key value, we stop the search, we have found what we were looking for!

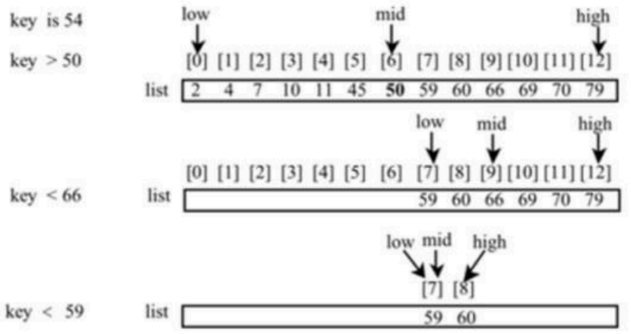
   Of course, we need to be keeping track of the indices of each of the elements, so we know what position to return, when we are all done.

   Here's an example of a binary search for 11 in the given list, that keeps track of index of the left-most element (i.e., the "lowest index" to consider), the index of the right-most element (i.e., the "highest index" to consider), and the index of the element in the "middle" of these two positions:



   The search method in the above example should then return a value of 4, the index of the found key value.

   Here's another example, where we are searching for a key value of 54:



Notice when we check if list[7] equals the key in the last step, and discover that 54 < 59, we must conclude that the key value -- if present -- would be left of list[7], in an area of the main list that was already eliminated. This, of course can't be -- so we know that key is not present in the list.

However, the position we found is still useful. We made an assumption with the binary search method that our list was in ascending order. For this to be true, one of two things must have happened -- either we sorted the list (which can be computationally expensive), or we never let our list get out of order in the first place. That is to say, every time we added a value to the list, we made sure we inserted it in just the right spot to preserve the order. Note, "just left" of list[7] is right where we would want to *insert*the key value of 54, Thus, we might have our search method not just return a negative value (like -1), to indicate the absence of the key value in the list, but also build into that negative value some useful information, like the index where the key value should be inserted.

Here's how this algorithm might look as code:

BinarySearch - Fast search in a sorted array

```
' Binary search in an array of any type
' Returns the index of the matching item, or -1 if the search fails
'
' The arrays *must* be sorted, in ascending or descending
' order (the routines finds out the sort direction).
' LASTEL is the index of the last item to be searched, and is
' useful if the array is only partially filled.
'
```

```
' Works with any kind of array, including objects if your are searching
' for their default property, and excluding UDTs and fixed-length strings.
' String are compared in case-sensitive mode.
'
' You can write faster procedures if you modify the first line
' to account for a specific data type, eg.
'   Function BinarySearchL (arr() As Long, search As Long,
'  Optional lastEl As Variant) As Long

Function BinarySearch(arr As Variant, search As Variant, _
    Optional lastEl As Variant) As Long
    Dim index As Long
    Dim first As Long
    Dim last As Long
    Dim middle As Long
    Dim inverseOrder As Boolean

    ' account for optional arguments
    If IsMissing(lastEl) Then lastEl = UBound(arr)

    first = LBound(arr)
    last = lastEl

    ' deduct direction of sorting
    inverseOrder = (arr(first) > arr(last))

    ' assume searches failed
    BinarySearch = first - 1

    Do
        middle = (first + last) \ 2
        If arr(middle) = search Then
            BinarySearch = middle
            Exit Do
        ElseIf ((arr(middle) < search) Xor inverseOrder) Then
            first = middle + 1
        Else
            last = middle - 1
        End If
    Loop Until first > last
End Function
```

Sorting Arrays

The binary search method, in both of the implementations discussed above required the array be pre-sorted. What if that's not the case? How can we take an unsorted array and sort it?

Bubblesort

Bubblesort is a specialized algorithm designed for sorting items that are already mostly in sorted order. If only one or two items in your list are out of order, bubblesort is very fast. If the items in your list are initially arranged randomly, bubblesort is extremely slow. For this reason you should be careful when you use bubblesort.

The idea behind the algorithm is to scan through the list looking for two adjacent items that are out of order. When you find two such items, you swap them and continue down the list. You repeat this process until all of the items are in order.

Figure 1 shows a small list where the item 1 is out of order. During the first pass through the list, the algorithm will find that items 4 and 1 are out of order so it swaps them. During the next pass it finds that items 3 and 1 are out of order so it swaps them. During the third pass it swaps items 2 and 1 and the list is in its final order. The way in which the item 1 seems to bubble towards the top of the list is what gives bubblesort its name.

2	2	2	1
3	3	1	2
4	1	3	3
1	4	4	4

Figure 1. In a bubblesort, item 1 slowly "bubbles" to the top.

You can improve the algorithm if you alternate upward and downward passes through the list. During downward passes an item that is too far down in the list, like the item 1 in the previous example, can move up only one position. An item that is too far up in the list might move many positions. If you add upward passes through the list, you will be able to move items many positions up through the list as well.

During each pass through the list, at least one new item reaches its final position. If the items in your list begin mostly in sorted order, the algorithm will need only one or two passes through the list to finish the ordering. If you have a list of 1,000 items with only one out of order, the algorithm would require only 2,000 steps to put the list in its proper order.

If the items begin arranged randomly, the algorithm may need one pass per item in the list. The algorithm would need up to 1 million steps to arrange a list of 1,000 items.

Listing 1 shows the VB code for the improved bubblesort algorithm.

```
' min and max are the minimum and maximum indexes
' of the items that might still be out of order.
Sub BubbleSort (List() As Long, ByVal min As Integer, _
    ByVal max As Integer)
```

```
Dim last_swap As Integer
Dim i As Integer
Dim j As Integer
Dim tmp As Long

    ' Repeat until we are done.
    Do While min < max
        ' Bubble up.
        last_swap = min - 1
        ' For i = min + 1 To max
        i = min + 1
        Do While i <= max
            ' Find a bubble.
            If List(i - 1) > List(i) Then
                ' See where to drop the bubble.
                tmp = List(i - 1)
                j = i
                Do
                    List(j - 1) = List(j)
                    j = j + 1
                    If j > max Then Exit Do
                Loop While List(j) < tmp
                List(j - 1) = tmp
                last_swap = j - 1
                i = j + 1
            Else
                i = i + 1
            End If
        Loop
        ' Update max.
        max = last_swap - 1

        ' Bubble down.
        last_swap = max + 1
        ' For i = max - 1 To min Step -1
        i = max - 1
        Do While i >= min
            ' Find a bubble.
            If List(i + 1) < List(i) Then
                ' See where to drop the bubble.
                tmp = List(i + 1)
                j = i
                Do
                    List(j + 1) = List(j)
                    j = j - 1
                    If j < min Then Exit Do
                Loop While List(j) > tmp
                List(j + 1) = tmp
                last_swap = j + 1
                i = j - 1
            Else
                i = i - 1
            End If
        Loop
        ' Update min.
        min = last_swap + 1
    Loop
End Sub
```

Listing 1. Bubblesort.

Selectionsort
Selectionsort is a very simple algorithm. First you search the list for the smallest item. Then you swap that item with the item at the top of the list. Next you find the second smallest item and swap it with the second item in the list. You continue finding the next smallest item and swapping it into its final position in the list until you have swapped all of the items to their final positions. The VB code for selectionsort is shown in Listing 2.

```
Sub Selectionsort (List() As Long, min As Integer, _
    max As Integer)
Dim i As Integer
Dim j As Integer
Dim best_value As Long
Dim best_j As Integer

    For i = min To max - 1
        best_value = List(i)
        best_j = i
        For j = i + 1 To max
            If List(j) < best_value Then
                best_value = List(j)
                best_j = j
            End If
        Next j
        List(best_j) = List(i)
        List(i) = best_value
    Next i
End Sub
```

Listing 2. Selectionsort.

While looking for the Ith smallest item, you must examine each of the N - I items that you have not yet placed in their final positions. Then the total number of steps the algorithm needs is:

```
    N + (N - 1) + (N - 2) + ... + 1 = N * (N + 1) / 2
```

This function is on the order of N2. That means if you increase the number of items in the list by a factor of 2, the run time of the algorithm will increase by a factor of roughly 22 = 4. There are several other sorting algorithms that require only about N * log(N) steps (quicksort is one described below), so selectionsort is not a very fast algorithm for large lists.

Selectionsort is fine for small lists, however. It is very simple so it is easy to program, debug, and maintain over time. In fact it is so simple that it is actually faster than the more complicated algorithms if the list you are sorting is very small. If your list contains only a dozen or so items, selectionsort will probably be your best choice.

Quicksort
Quicksort is a recursive algorithm that uses a divide-and-conquer technique. While the list of items to be sorted contains at least two items, quicksort divides it into two sublists and recursively calls itself to sort the sublists.

The quicksort routine first checks to see if the list it is sorting contains fewer than two items. If so, it simply returns.

Otherwise the subroutine picks an item from the list to use as a dividing point. It then places all of the items that belong before this dividing point in the left part of the list. It places all of the other items in right part of the list. The subroutine then recursively calls itself to sort two smaller sublists.

There are several ways in which the quicksort routine might pick the dividing item. One of the easiest is to simply use the first item in the sublist being sorted. If the list is initially arranged randomly, that item will be a reasonable choice. Chances are good that the item will belong somewhere in the middle of the list and the two sublists the algorithm creates will be reasonably equal in size.

If the numbers are initially sorted or almost sorted, or if they are initially sorted in reverse order, then this method fails miserably. In that case the first item in the list will divide the list into one sublist that contains almost every item and another that will contain almost no items. Since the larger sublist does not shrink much, the algorithm makes little headway. In this case the algorithm will require on the order of N2 steps. This is the same order of performance given by selectionsort, only this algorithm is much more complicated.

A better method for selecting the dividing item is to choose one randomly. Then no matter how the items in the list are arranged, chances are the item you select will belong near the middle of the list and the sublists will be fairly evenly sized.

As long as the sublists are fairly equal in size, the algorithm will require on the order of N * log(N) steps. It can be proven that this is the fastest time possible for a sorting algorithm that sorts using comparisons. By using a little randomness, this algorithm avoids the possibility of its worst case N2 behavior and gives an expected case performance of N * log(N). Quicksort is very fast in practice as well as theory, so it is the favorite sorting algorithm of many programmers.

Listing 2 shows the VB code for the quicksort routine.

```
Sub Quicksort (List() As Long, min As Integer, max As Integer)
Dim med_value As Long
Dim hi As Integer
Dim lo As Integer
Dim i As Integer
```

```
    ' If the list has no more than 1 element, it's sorted.
    If min >= max Then Exit Sub

    ' Pick a dividing item.
    i = Int((max - min + 1) * Rnd + min)
    med_value = List(i)

    ' Swap it to the front so we can find it easily.
    List(i) = List(min)

    ' Move the items smaller than this into the left
    ' half of the list. Move the others into the right.
    lo = min
    hi = max
    Do
        ' Look down from hi for a value < med_value.
        Do While List(hi) >= med_value
            hi = hi - 1
            If hi <= lo Then Exit Do
        Loop
        If hi <= lo Then
            List(lo) = med_value
            Exit Do
        End If

        ' Swap the lo and hi values.
        List(lo) = List(hi)

        ' Look up from lo for a value >= med_value.
        lo = lo + 1
        Do While List(lo) < med_value
            lo = lo + 1
            If lo >= hi Then Exit Do
        Loop
        If lo >= hi Then
            lo = hi
            List(hi) = med_value
            Exit Do
        End If

        ' Swap the lo and hi values.
        List(hi) = List(lo)
    Loop

    ' Sort the two sublists
    Quicksort List(), min, lo - 1
    Quicksort List(), lo + 1, max
End Sub
```

Listing 3. Quicksort.

Countingsort

The discussion of quicksort above mentions that the fastest possible sorting algorithms that use comparisons use on the order of N * log(N) steps. Countingsort does not use comparisons so it is not bound by that result. In fact countingsort is so fast it seems to sort using magic rather than comparisons.

On the other hand, countingsort only works under special circumstances. First, the items you are sorting must be integers. You cannot use countingsort to sort strings. Second, the range of values the items have must be fairly limited. If your items range from 1 to 1,000, countingsort will work extremely well. If the items range from 1 to 30,000, countingsort will not work as well. If the items range from 1 to 10 billion, you should go back to quicksort.

The algorithm starts by allocating a temporary array of integers with bounds that cover the range of your items. If your items range from min_item to max_item, the algorithm creates an array like this:

```
Dim Counts() As Integer

ReDim Counts(min_item To max_item)
```

Next the algorithm looks through each of the items in the list and increments the Counts entry corresponding to that item. When this stage is finished, Counts(I) holds a count of the number of items that have value I. Keep in mind that the ReDim statement initializes each of these entries to 0 so you do not need to do this yourself.

```
For I = min To Max
    Counts(List(I)) = Counts(List(I)) + 1
Next I
```

The program then runs through the Counts array converting the counts into offsets in the sorted list. For example, suppose the items in the list have values between 1 and 1,000. There might be 15 items with value 1, 7 with value 2, 11 with value 3, and so forth. The items with value 1 will begin at position 1 in the sorted list. Since there are 15 of them, the items with value 2 will start at position 16. There are 7 of those so the items with value 3 will start at position 23, etc.

```
next_spot = 1
For i = min_value To max_value
    this_count = counts(i)
    counts(i) = next_spot
    next_spot = next_spot + this_count
Next i
```

When this stage is complete, the entry Counts(I) indicates the position in the sorted list where the first item with value I belongs.

Now the algorithm reads through the list again, placing each item in the correct position in the sorted array. As the algorithm places each item, it updates the corresponding Counts entry so the next item with the same value goes into the next position in the array.

If you are sorting data records rather than just numbers, you will need to use a temporary array. Since you place each item directly in its final location in the sorted array, you cannot store the sorted list in the same array as the original list. If you did, you would overwrite another item in the list that might not yet have been moved to its correct location. If your program needs the items in the original array, you will have to copy them back out of the temporary array when you have finished sorting them.

To sort N numbers that have a range that spans M values, countingsort executes roughly 2 * N + M steps. First it reads the N items to fill in the Counts array. Then it runs through the M values in the Counts array converting them from counts into offsets. Finally it moves the N items to their correct sorted positions.

If N is large and M is relatively small, 2 * N + M is much faster than the N * log(N) performance given by quicksort. To sort 30,000 numbers that ranged from 1 to 10,000, for example, quicksort might execute more than 400,000 steps. Countingsort would execute only about 70,000 steps and take less than a fifth as long.

The VB code for countingsort is shown in Listing 4.

```
Sub Countingsort (List() As Long, sorted_list() As Long, _
    min As Integer, max As Integer, min_value As Long, _
    max_value As Long)
Dim counts() As Integer
Dim i As Integer
Dim this_count As Integer
Dim next_offset As Integer

    ' Create the Counts array.
    ReDim counts(min_value To max_value)

    ' Count the items.
    For i = min To max
        counts(List(i)) = counts(List(i)) + 1
    Next i

    ' Convert the counts into offsets.
    next_offset = min
    For i = min_value To max_value
        this_count = counts(i)
        counts(i) = next_offset
        next_offset = next_offset + this_count
    Next i

    ' Place the items in the sorted array.
    For i = min To max
        sorted_list(counts(List(i))) = List(i)
        counts(List(i)) = counts(List(i)) + 1
    Next i
End Sub
```

Listing 4. Countingsort.

Summary
Table 1 summarizes the strengths and weaknesses of the algorithms presented here. As you can see, each performs well under some circumstances and badly under others. Here are some guidelines to help you select the right algorithm for your situation.

- If your list is more than 99% sorted already, use bubblesort.
- If you have a very small list (under 100 items or so), use selectionsort.
- If the items in your list are integers ranging over a small number of values (up to several thousand), use countingsort.
- Otherwise use quicksort.

Algorithm	Advantages	Disadvantages
Bubblesort	Very fast for lists that are almost sorted	Very slow sorting all other lists
Selectionsort	Very simple	Slow for large lists
	Easy to understand	
	Very fast for small lists	
Quicksort	Very fast for large lists	Trouble if there are lots of duplicate data values
Countingsort	Extremely fast when the data is distributed over a small range (e.g. all values are between 1 and 1,000)	Slower when data range is large
		Requires extra memory
		Only works with integer data

Table 1. Comparison of Sorting Algorithms.

Program SORT allows you to run each of these algorithms. You can enter the number of items to sort and the maximum value the items should have. When you press the Go button, the program builds a list of random numbers between 1 and the maximum you specified. The program then sorts the numbers and displays the amount of time it took.

If you check the Start Sorted check box, the program will begin by sorting the list using Quicksort. If you also fill in a value in the # Unsorted field, the program will then switch some items around so the list contains the indicated number of unsorted items. The program will then sort the partially sorted list using the algorithm you have selected. This allows you to test bubblesort on lists that are mostly sorted.

Finally, you can fill in a value in the Repetitions field to make the program sort the list many repeatedly. You will need to sort small lists many times to see that selectionsort is faster than quicksort and countingsort for very small lists.

Using the program you can verify that:

- Selectionsort is fastest for small lists (use many repetitions).
- Quicksort is faster for larger lists.
- Countingsort is faster still if the number of items is larger than the largest item.
- Quicksort is faster than countingsort if the largest item is large (like 30,000) compared to the number of items (like 1,000).
- Bubblesort is fastest if the list starts mostly sorted.

CHAPTER 9: LINKING TO DATABASES

Database controls

Using **Database Control**, you can perform administrative tasks such as creating schema objects (tables, views, indexes, and so on), managing user security, managing **database** memory and storage, backing up and recovering your **database**, and importing and exporting data.

Data control and data-bound control

Data control	**Data-bound**
Using data control is a two-step process. First you place a data control on a form and set the properties to link it to a database file and table	You create the controls, such as labels and text boxes, to display the actual data. Each control is a bound to particular field in the table. In this example the label is called a data bound control and automatically displays the contents of bound field when the project runs.
Data control generally links one form with one table.	If you want to have data-bound controls on second form, you must place a data control on that form.
Prefix of data control is “dat”	For data-bound control prefix depends upon the control which you are using.
Data control to work you need to set some properties to connect with database like connect property, database name property, record source property.	To display data on the data-bound control that you are using like labels or textboxes. You need set it?s data source property and data field name which is column name form the table.

Using the ADO Data Control

The ADO Data control uses Microsoft ActiveX Data Objects (ADO) to quickly create connections between data-bound controls and data providers. Data-bound controls are any controls that feature a DataSource property. Data providers can be any source written to the OLE DB specification. You can also easily create your own data provider using Visual Basic's class module.

Although you can use the ActiveX Data Objects directly in your applications, the ADO Data control has the advantage of being a graphic control (with Back and Forward buttons) and an easy-to-use interface that allows you to create database applications with a minimum of code.

Figure 7.4 The ADO Data Control

Several of the controls found in Visual Basic's Toolbox can be data-bound, including the CheckBox, ComboBox, Image, Label, ListBox, PictureBox, and TextBox controls. Additionally, Visual Basic includes several data-bound ActiveX controls such as the DataGrid, DataCombo, Chart, and DataList controls. You can also create your own data-bound ActiveX controls, or purchase controls from other vendors.

Previous versions of Visual Basic featured the intrinsic Data control and the Remote Data control (RDC) for data access. Both controls are still included with Visual Basic for backward compatibility. However, because of the flexibility of ADO, it's recommended that new database applications be created using the ADO Data Control.

For More Information A complete list of data-bound controls can be found in "Controls That Bind to the ADO Data Control." To find out how to use the intrinsic Data control or the Remote Data control, see "Using the Data Control" or "Using the Remote Data Control." For details about creating a data provider, see "Creating Data-Aware Classes."

Possible Uses

- Connect to a local or remote database.
- Open a specified database table or define a set of records based on a Structured Query Language (SQL) query or stored procedure or view of the tables in that database.
- Pass data field values to data-bound controls, where you can display or change the values.
- Add new records or update a database based on any changes you make to data displayed in the bound controls.

To create a client, or front-end database application, add the ADO Data control to your forms just as you would any other Visual Basic control. You can have as many ADO Data controls on your form as you need. Be aware, however, that the control is a comparatively "expensive" method of creating connections, using at least two connections for the first control, and one more for each subsequent control.

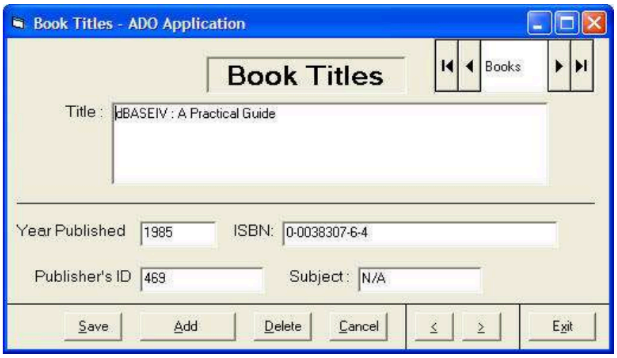
Active Data Object(ADO) Connection

Since data control is not a very flexible tool as it only work with limited kinds of data and must work strictly in the Visual Basic environment.

To overcome these limitations, we can use a much more powerful data control in Visual Basic, known as ADO control. ADO stands for ActiveX data objects. As ADO is ActiveX-based, it can work in different platforms (different computer systems) and different programming languages. Besides, it can access many different kinds of data such as data displayed in the Internet browsers, email text and even graphics other than the usual relational and non-relational database information.

To be able to use ADO data control, you need to insert it into the toolbox. To do this, simply press Ctrl+T to open the components dialog box and select Microsoft ActiveX Data Control 6. After this, you can proceed to build your ADO-based VB database applications.

The following example will illustrate how to build a relatively powerful database application using ADO data control. First of all, name the new form as **frmBookTitle** and change its caption to **Book Titles- ADO Application**. Secondly, insert the ADO data control and name it as **adoBooks** and change its caption to **book**. Next, insert the necessary labels, text boxes and command buttons. The runtime interface of this program is shown in the diagram below, it allows adding and deletion as well as updating and browsing of data.



Title : dBASEIV : A Practical Guide

Year Published 1985 ISBN: 0-0038307-6-4

Publisher's ID 469 Subject : N/A

The property settings of all the controls are listed as follow:

Control Property	**Setting**
Form Name	frmBookTitle
Form Caption	Book Titles - ADOApplication
ADO Name	adoBooks
Label1 Name	lblApp
Label1 Caption	Book Titles
Label 2 Name	lblTitle
Label2 Caption	Title :
Label3 Name	lblYear
Label3 Caption	Year Published:
Label4 Name	lblISBN
Label4 Caption	ISBN:

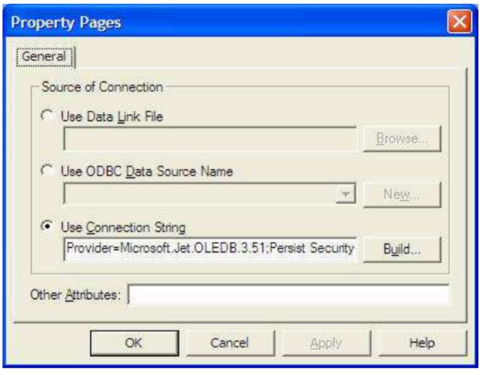
Labe5 Name	lblPubID
Label5 Caption	Publisher's ID:
Label6 Name	lblSubject
Label6 Caption	Subject :
TextBox1 Name	txtitle
TextBox1 DataField	Title
TextBox1 DataSource	adoBooks
TextBox2 Name	txtPub
TextBox2 DataField	Year Published
TextBox2 DataSource	adoBooks
TextBox3 Name	txtISBN
TextBox3 DataField	ISBN
TextBox3 DataSource	adoBooks
TextBox4 Name	txtPubID
TextBox4 DataField	PubID
TextBox4 DataSource	adoBooks
TextBox5 Name	txtSubject
TextBox5 DataField	Subject
TextBox5 DataSource	adoBooks
Command Button1 Name	cmdSave
Command Button1 Caption	&Save
Command Button2 Name	cmdAdd
Command Button2 Caption	&Add
Command Button3 Name	cmdDelete
Command Button3 Caption	&Delete
Command Button4 Name	cmdCancel
Command Button4 Caption	&Cancel
Command Button5 Name	cmdPrev
Command Button5 Caption	&<

Command Button6 Name	cmdNext
Command Button6 Caption	&>
Command Button7 Name	cmdExit
Command Button7 Caption	E&xit

To be able to access and manage a database, you need to connect the ADO data control to a database file. We are going to use **BIBLIO.MDB** that comes with VB6. To connect ADO to this database file , follow the steps below:

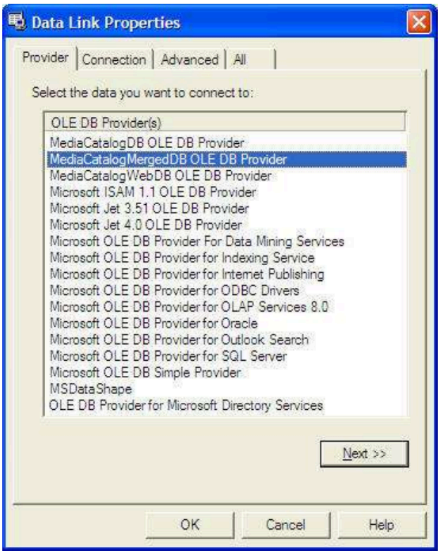
a) Click on the ADO control on the form and open up the properties window.

b) Click on the ConnectionString property, the following dialog box will appear.



Provider=Microsoft.Jet.OLEDB.3.51;Persist Security

When the dialog box appear, select the Use **Connection String**'s Option. Next, click build and at the **Data Link dialog box**, double-Click the option labeled **Microsoft Jet 3.51 OLE DB provider.**


Data Link Properties

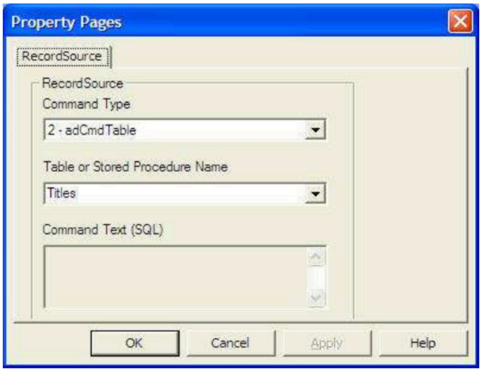
Provider | Connection | Advanced | All

Select the data you want to connect to:

OLE DB Provider(s)
MediaCatalogDB OLE DB Provider
MediaCatalogMergedDB OLE DB Provider
MediaCatalogWebDB OLE DB Provider
Microsoft ISAM 1.1 OLE DB Provider
Microsoft Jet 3.51 OLE DB Provider
Microsoft Jet 4.0 OLE DB Provider
Microsoft OLE DB Provider For Data Mining Services
Microsoft OLE DB Provider for Indexing Service
Microsoft OLE DB Provider for Internet Publishing
Microsoft OLE DB Provider for ODBC Drivers
Microsoft OLE DB Provider for OLAP Services 8.0
Microsoft OLE DB Provider for Oracle
Microsoft OLE DB Provider for Outlook Search
Microsoft OLE DB Provider for SQL Server
Microsoft OLE DB Simple Provider
MSDataShape
OLE DB Provider for Microsoft Directory Services

After that, click the Next button to select the file **BIBLO.MDB.** You can click on Text Connection to ensure proper connection of the database file. Click OK to finish the connection.

Finally, click on the RecordSource property and set the **command type** to **adCmd Table** and **Table name** to **Titles**. Now you are ready to use the database file.



RecordSource
Command Type: 2 - adCmdTable
Table or Stored Procedure Name: Titles
Command Text (SQL)

Now, you need to write code for all the command buttons. After which, you can make the ADO control invisible.

For the **Save** button, the program codes are as follow:

```
Private Sub cmdSave_Click()

adoBooks.Recordset.Fields("Title") = txtTitle.Text
adoBooks.Recordset.Fields("Year Published") = txtPub.Text
adoBooks.Recordset.Fields("ISBN") = txtISBN.Text
adoBooks.Recordset.Fields("PubID") = txtPubID.Text
adoBooks.Recordset.Fields("Subject") = txtSubject.Text
adoBooks.Recordset.Update

End Sub
```

For the **Add** button, the program codes are as follow:

```
Private Sub cmdAdd_Click()

adoBooks.Recordset.AddNew

End Sub
```

For the **Delete** button, the program codes are as follow:

```
Private Sub cmdDelete_Click()
```

```
Confirm = MsgBox("Are you sure you want to delete this record?", vbYesNo, "Deletion
Confirmation")
If Confirm = vbYes Then
adoBooks.Recordset.Delete
MsgBox "Record Deleted!", , "Message"
Else
MsgBox "Record Not Deleted!", , "Message"
End If

End Sub
```

For the **Cancel** button, the program codes are as follow:

```
Private Sub cmdCancel_Click()

txtTitle.Text = ""
txtPub.Text = ""
txtPubID.Text = ""
txtISBN.Text = ""
txtSubject.Text = ""

End Sub
```

For the Previous (<) button, the program codes are

```
Private Sub cmdPrev_Click()


If Not adoBooks.Recordset.BOF Then
adoBooks.Recordset.MovePrevious
If adoBooks.Recordset.BOF Then
adoBooks.Recordset.MoveNext
End If
End If



End Sub
```

For the Next(>) button, the program codes are

```
Private Sub cmdNext_Click()


If Not adoBooks.Recordset.EOF Then
```

```
adoBooks.Recordset.MoveNext
If adoBooks.Recordset.EOF Then
adoBooks.Recordset.MovePrevious
End If
End If

End Sub
```

Database Reports

Once the Data Environment designer has been created, you can create a data report. Because not all of the fields in the data environment will be useful in a report, this series of topics creates a limited report that displays only a few fields.

To create a new data report

1. On the **Project** menu, click **Add Data Report**, and Visual Basic will add it to your project. If the designer is not on the **Project** menu, click **Components**. Click the **Designers** tab, and click **Data Report** to add the designer to the menu.

   Note The first four kinds of ActiveX designers loaded for a project are listed on the **Project** menu. If more than four designers are loaded, the later ones will be available from the **More ActiveX Designers** submenu on the **Project** menu.

2. Set the properties of the **DataReport** object according to the table below:

Property	**Setting**
Name	rptNwind
Caption	Northwind Data Report

3. On the **Properties** window, click **DataSource** and then click **deNwind**. Then click **DataMember** and click **Customers**.

   Important To set the DataSource property to deNwind, the Data Environment designer must be open. If it is closed, press CTRL+R to display the Project window, then double-click the data environment icon.

4. Right-click the Data Report designer, and click **Retrieve Structure**.

   You have added a new group section to the designer. Each group section has a one-to-one correspondence to a Command object in the data environment; in this case, the new Group section corresponds to the Customers Command object. Notice also that the Group Header has a matching Group Footer section.

Note The Data Environment allows you to create hierarchies of Command objects wherein a Command object has more than one child object — child Command objects parallel to each other. The Data Report designer, however, is not as flexible, and can't display more than one child object at a time. In such cases, when executing a Retrieve Structure command, the Data Report will display only the first of the child commands, and none below it. Thus you should avoid creating Command hierarchies with parallel children commands.

5. From the Data Environment designer, drag the **CompanyName** field (under the **Customers** command) onto the **Group Header (Customers_Header)** section.

   The Group Header section can contain any field from the Customers command, however, for demonstration purposes, only the Customer name is displayed at this time.

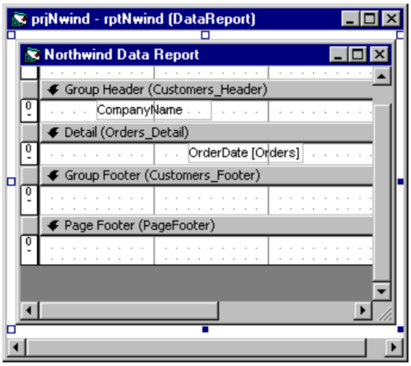
6. Delete the Label control (rptLabel) named **Label1**.

   If you do not want a Label control to be included with the TextBox control, you can uncheck the **Drag and Drop Fields Caption** option on the **Field Mapping** tab of the Data Environment designer's **Options** dialog box.

7. From the Data Environment designer, drag the **OrderDate** field (under the **Orders** command) onto the **Details (Orders_Detail)** section. Delete the Label control.

   The Details section represents the innermost "repeating" section, and thus corresponds to the lowest Command object in the Data Environment hierarchy: the Orders Command object.

8. Resize the Data Report designer's sections to resemble the figure below:



It's important to resize the height of the Details section to be as short as possible because the height will be multiplied for every OrderDate returned for the CompanyName. Any extra space below or above the OrderDate text box will result in unneeded space in the final report.

9. Save the project.

Preview the Data Report Using the Show Method

Now that the data environment and the data report objects have been created, you are almost ready to run the project. One step remains: to write code to show the data report.

To show the data report at run time

1. On the **Project Explorer** window, double-click the **frmShowReport** icon to display the Form designer.

2. On **Toolbox**, click the **General** tab.

   When you add a Data Report designer to your project, its controls are added to the tab named **DataReport**. To use the standard Visual Basic controls, you must switch to the General tab.

3. Click the **CommandButton** icon and draw a **CommandButton** on the form.

4. Set the properties of the **Command1** control according to the table below:

Property	**Setting**
Name	cmdShow
Caption	Show Report

5. In the button's Click event, paste the code below.

```
6. Private Sub cmdShow_Click()
7.     rptNwind.Show
   End Sub
```

8. Save and run the project.

9. Click **Show Report** to display the report in print preview mode.

Optional—Setting the Data Report as the Startup Object

You can also display the data report with no code at all.

1. On the **Project** menu, click **prjNwind Properties**.

2. In the **Startup Object** box, select **rptNwind**.
3. Save and run the project.

   Note If you use this method, you can remove the Form object from your project.

CHAPTER 10: EMERGING TRENDS IN VISUAL PROGRAMMING

Emerging trends in visual programming

It wasn't many years ago that using a computer meant literally learning a new language.

Innovations like the graphical user interface exposed basic elements like the filesystem to a wider audience, and the Internet has become increasingly democratized as user-friendly tools like Wordpress, Youtube and Soundcloud allow anyone to create, publish and distribute content without writing a line of code. Today an explosion of accessible prototyping kits is making it possible for amateurs and hobbyists to sink their teeth into the growing Internet of Things by cobbling together connected computing projects.

But when it comes to making that hardware do your bidding, most tinkerers will still encounter a "language barrier". Even the most user-friendly development boards need to be programmed; and even the simplest programming languages still look like alphabet soup to the uninitiated.

Fortunately, some developers have started to step in and provide user-friendly, visual programming tools. These platforms abstract away the functions, variables and idiosyncratic syntax rules of the underlying code and give users a simple drag-and-drop interface for building apps out of discrete chunks of logic ("When this happens, do that") and widgets that can apply settings tailored to any specific piece of hardware.

Here's a roundup of visual programming platforms that have some application to the Internet of Things—either because they're tailor-made for programming sensors and embedded computers, or because they're general-purpose platforms for programming in the languages that are compatible with those devices.

- Free / Open Source
- Commercial /Enterprise Focused
- Additional Tools and Resources

Challenges of emerging trends in Visual programming

Visual languages have a fundamental set of problems:

1. Visual Languages Aren't Extensible

This is probably the capital sin of visual languages. They allow you to do a limited set of things easily, but edge cases are far too difficult, or even impossible to achieve. Tools should give us more power, instead of limiting us.

2. Visual Languages Generate Slow Code

Every developer who has faced performance problems knows how hard they are to diagnose and overcome. Visual languages are leaky abstractions, often generating slow code which is impossible to optimize.

3. Visual Language Tools Can Be Terrible

We live and breathe in our IDEs (Integrated Development Environments). When they are poor, they can make our lives miserable! Visual languages and IDEs should be designed together: our love or hate for a language is a direct measure of our love or hate for its tools.

4. Visual Languages Lock You In

Any technology decision brings a level of lock in. The fear of being locked into a dead-end is justifiable, given that most visual languages generate unreadable lower level code, only target niche segments, are supported by suicidal startups, or haven't left the research lab yet (where amazing recent work, like Bret Victor's, is being done). Real success stories, of people using it in large projects, are still rare, and these communities are still growing.

5. You Are Neurologically Programmed to Reject It

The problems in the previous sections are serious and, as visual language architects, we know it's our responsibility to continue our work to address them, removing these limitations.

But there are deeper reasons why you don't trust visual languages:

The first is related to our love for complexity. Although we might say otherwise, we all do, according to the father of usability, Don Norman. Take musical instruments, for instance: musicians love them because they are hard to master. Think about legislation. Or even language: most poetry is hidden under complex sentences and words. We love the intellectual challenge of thinking difficult thoughts. Your mind is probably doing it right now, agreeing with these ideas, or checking if there is anything wrong or missing in them.

We need, however, to be careful with this, as we create deep moats against the rest of the world, and that brilliant xor swap algorithm I've coded today may be somebody else's nightmare next year.

Like Steve Jobs said: "Simple can be harder than complex. But it's worth it."

The second reason is even more primitive: it's fear of change. We take the time to acquire some expertise, and suddenly it seems our old weapons are no longer needed. But we're seeing it wrong:

our weapons are not to be able to do a malloc, or pointer arithmetic; they are to be able to divide a problem into smaller parts, to understand the process of iterating on its solution, to detect strange code smells, to discuss a diagram on a whiteboard, to refactor systems into better architectures.

Those are our real weapons, the ones we've been sharpening for years, weapons that, with higher level languages, become even more deadly to any problem that might cross our path. Fear not: a great developer will always be more valuable than any tool he works with.

Coping with challenges of emerging trends in visual programming

Let's face it: there's also social acceptance. We were here first, we bought the debut albums, we'd never be caught listening to the sellouts that make it easy for everyone.

1. We've Been Here Before

Because of these instinctive reactions, anything that aims to democratize computing is met with skepticism. HyperCard, Delphi, Visual Basic, COBOL. We, the "real" developers, are too quick to point out their faults, and mock them with no mercy until they die, instead of helping them improve. Even JavaScript almost had that same fate during the DHTML days.

There is, however, an analogy that might better show us the future: The shift from text operating systems, like Unix and MS-DOS, to graphical user interfaces, like Mac or Windows, which suffered pushback from several computer experts at the time.

That evolution, which seems so inevitable today, brought us better graphic cards – which lead to better games, brought us the world wide web, brought us smartphones.

Can you imagine a world without these unbelievable things?

2. So, Are We Finally Ready For It?

Not sure: you may not, or you may feel we're not – it's fair either way. But try to understand your biases, and don't be afraid to give visual languages a shot: there are several organizations like mine, in different domains, challenging the way software is traditionally delivered, and pushing forward the state of the art.

And even if, after trying them out, you still don't see the bright future of visual programming, that's fine: consider keeping an open mind and come back later, otherwise you might just end up in the wrong side of history.