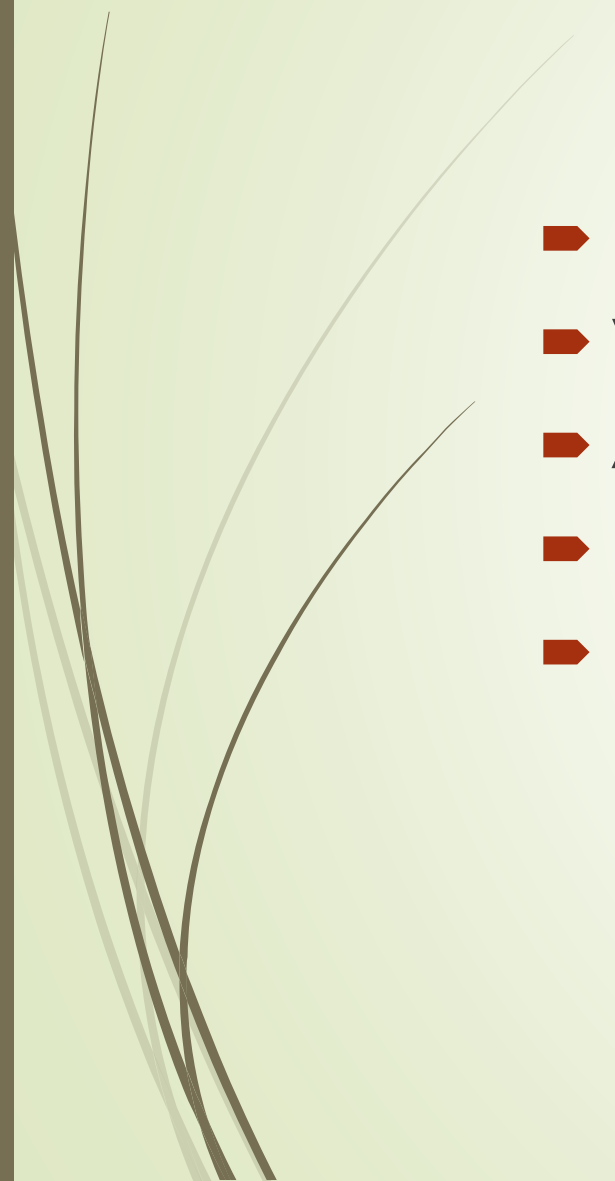




# Visual Basic.NET Introduction



# ROAD MAP

- Introduction
  - Variables, Data Types, Constants
  - Arrays
  - Flow-Control Statements
  - Loop Structures
- 



# INTRODUCTION



- Visual basics and Visual Basics .NET
  - We've already learnt Visual basics
  - .NET : a name for a new software platform
  - .NET framework: fundamental component.
    - Enormous collection of functions for just about any programming task.
- Advantage:
  - Easy to learn
  - Powerful



# INTRODUCTION



- Visual basics and Visual Basics .NET
  - We've already learnt Visual basics
  - .NET : a name for a new strategy
  - .NET framework: fundamental component.
    - Enormous collection of functions for just about any programming task.
- Advantages:
  - Easy to learn
  - Powerful
  - Similar to C#

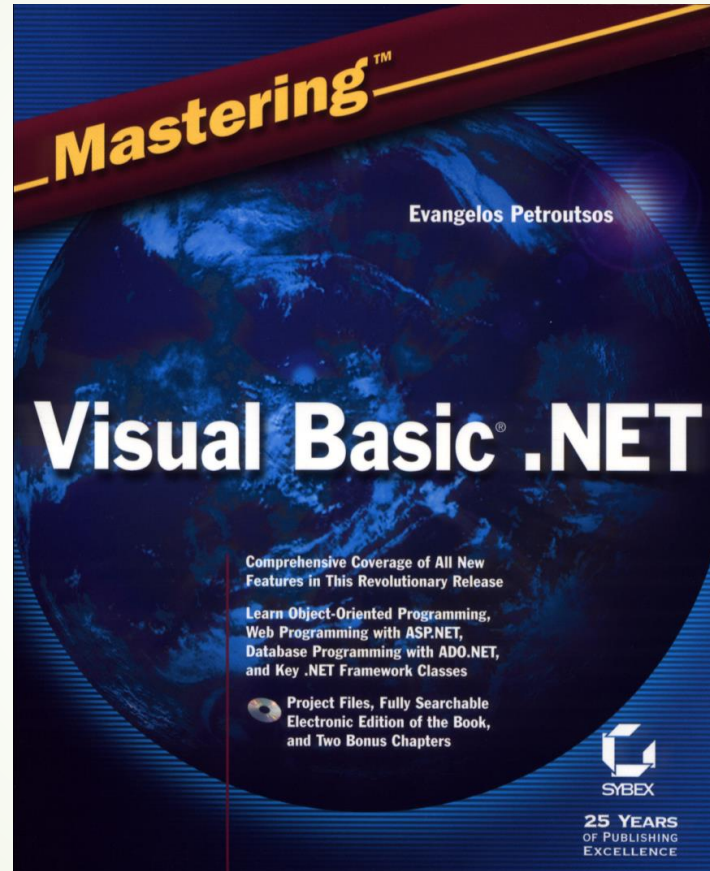


# Module Structure

- Topic 1 Visual Basic.NET Introduction
- Topic 2 Using Procedures
- Topic 3 Working with Forms
- Topic 4 Windows Controls
- Topic 5 More Windows Controls
- Topic 6 Building Custom Classes
- Topic 7 Building Custom Windows Controls
- Topic 8 Automation of Microsoft Office Applications

# Books

- Mastering Visual Basic .NET 2002
- A programmer's introduction to VB.NET 2001



Craig Utley

## A Programmer's Introduction to Visual Basic.NET

**SAMS**

201 West 103rd Street  
Indianapolis, IN 46290 USA



# Installation

- Environment: Visual Studio
- Microsoft official website



Visual  
Studio



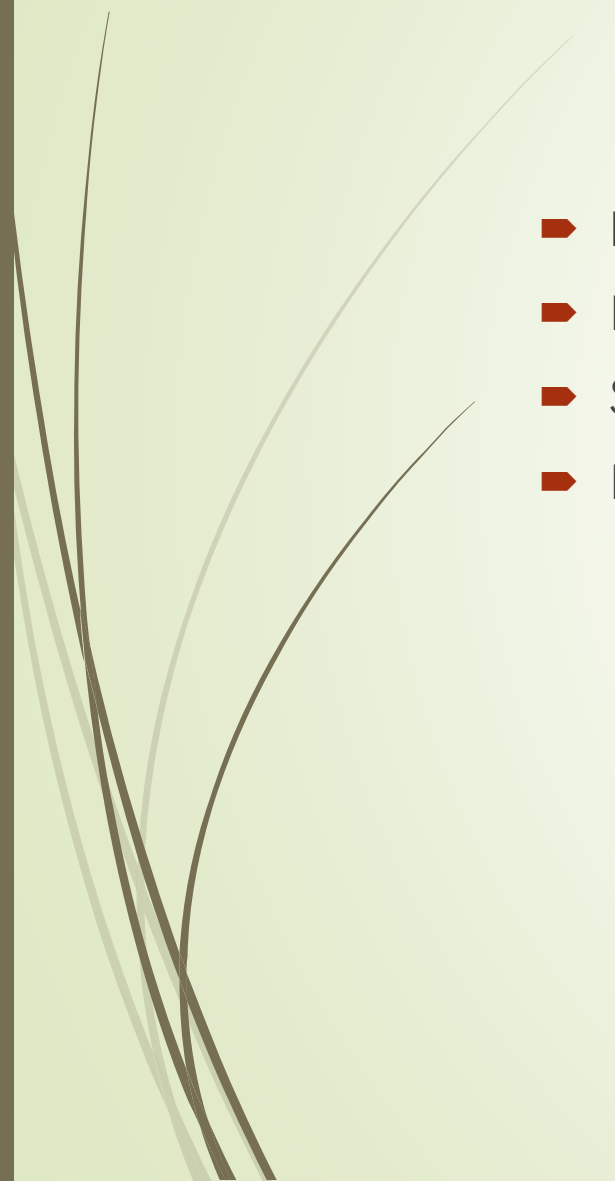
# Variables, Data Types and Constants

- Although not an absolute requirement, VB.NET encourages the declaration of variables, which is a default setting.
- Types of Variables
  - Numeric
  - String
  - Boolean
  - Date
  - Object





# Numeric Variables

- Integers (there are several integer data types)
  - Decimals
  - Single, or floating-point numbers with limited precision
  - Double, or floating-point numbers with extreme precision
- 


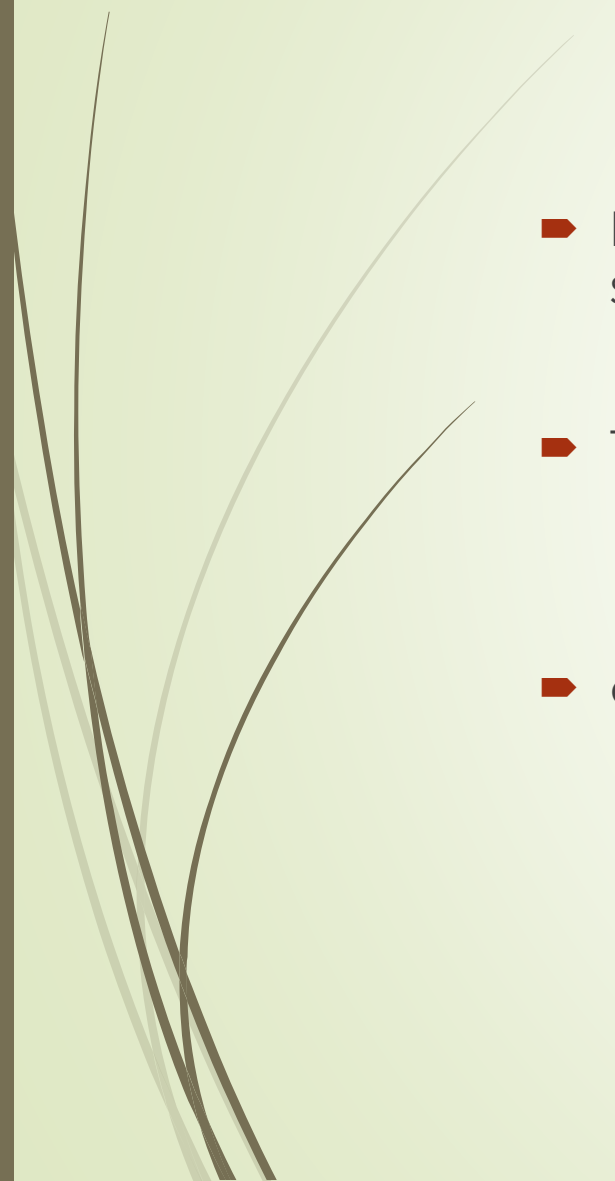
# Basic Numeric Data Type


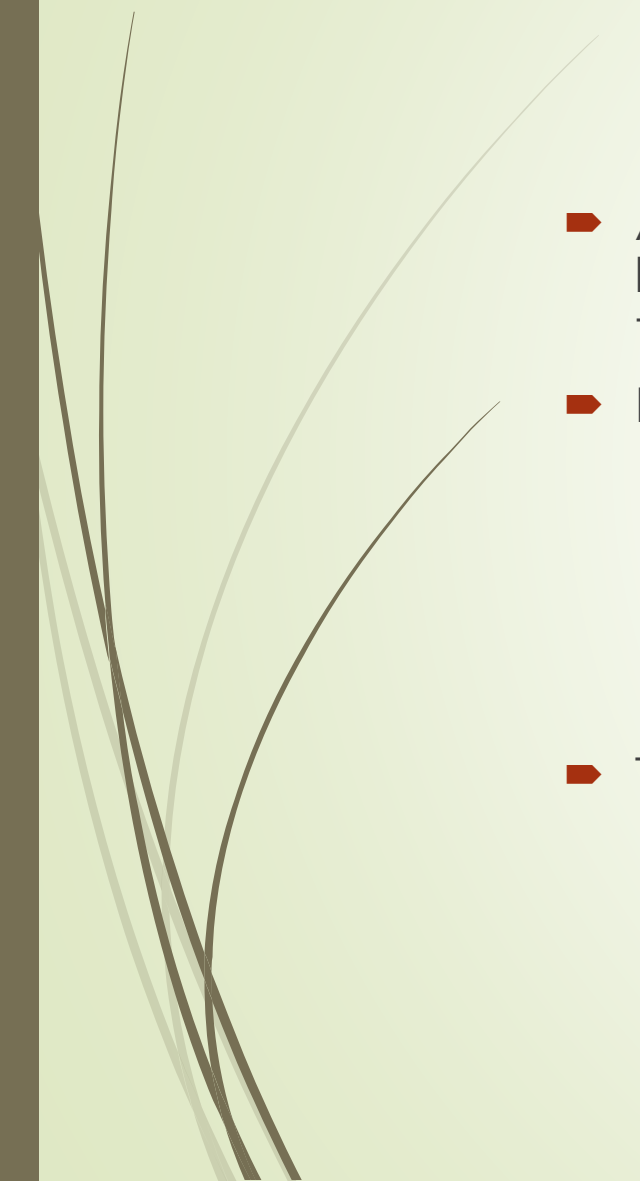
DATA TYPE	MEMORY REPRESENTATION	STORES
Short (Int16)	2 bytes	Integer values in the range $-32,768$ to $32,767$ .
Integer (Int32)	4 bytes	Integer values in the range $-2,147,483,648$ to $2,147,483,647$ .
Long (Int64)	8 bytes	Very large integer values.
Single	4 bytes	Single-precision floating-point numbers. It can represent negative numbers in the range $-3.402823E38$ to $-1.401298E-45$ and positive numbers in the range $1.401298E-45$ to $3.402823E38$ . The value 0 can't be represented precisely (it's a very, very small number, but not exactly 0).
Double	8 bytes	Double-precision floating-point numbers. It can represent negative numbers in the range $-1.79769313486232E308$ to $-4.94065645841247E-324$ and positive numbers in the range $4.94065645841247E-324$ to $1.79769313486232E308$ .
Decimal	16 bytes	Integer and floating-point numbers scaled by a factor in the range from 0 to 28. See the description of the Decimal data type for the range of values you can store in it.



# Infinity and Other Oddities

- VB.NET can represent two very special values, which may not be numeric values themselves but are produced by numeric calculations:
  - NaN (not a number) and Infinity.
- If your calculations produce NaN or Infinity, you should confirm the data and repeat the calculations, or give up.
- For all practical purposes, neither NaN nor Infinity can be used in everyday business calculations.

- 
- 
- Let's generate an Infinity value. Start by declaring a Double variable, The string "Infinity" will appear on a message box:
    - `Dim dblVar As Double = 999`
  - Then divide this value by zero:
    - `Dim infVar as Double`
    - `infVar = dblVar / 0`
  - and display the variable's value:
    - `MsgBox(infVar)`



- 
- 
- ▶ Another calculation that will yield a non-number is when you divide a very large number by a very small number. If the result exceeds the largest value that can be represented with the Double data type, the result is Infinity.
  - ▶ Declare three variables as follows:
    - ▶ `Dim largeVar As Double = 1E299`
    - ▶ `Dim smallVar As Double = 1E-299`
    - ▶ `Dim result As Double`
  - ▶ Then divide the large variable by the small variable and display the result:
    - ▶ `result = largeVar / smallVar`
    - ▶ `MsgBox(result)`



# Not a Number (NaN)

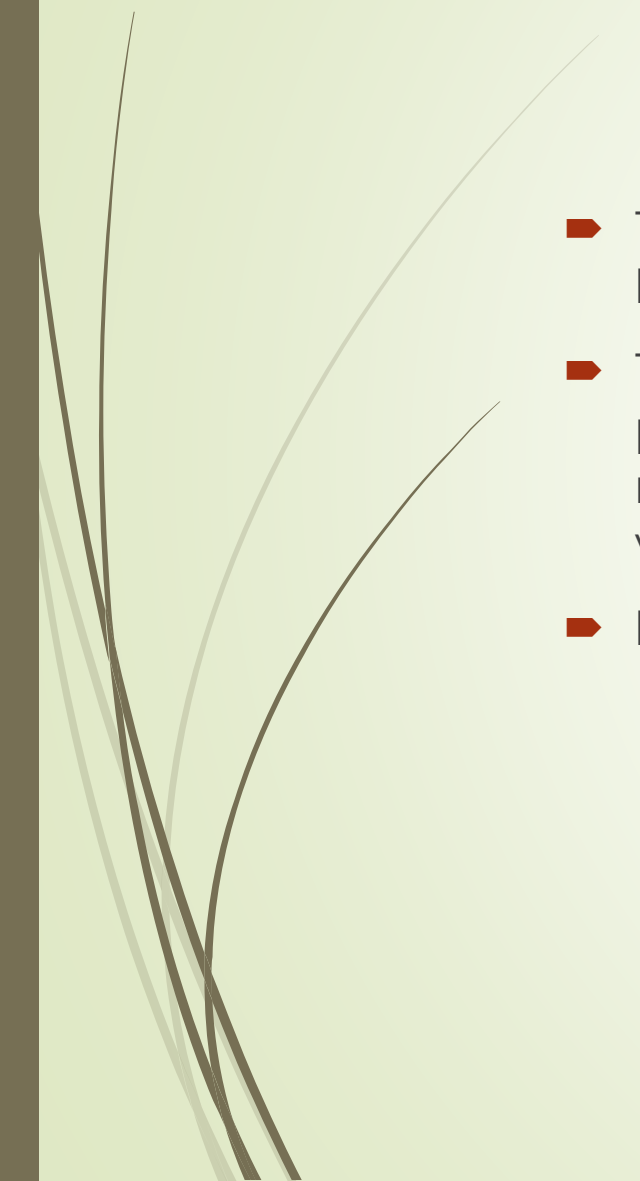
- ▶ NaN is not new. Packages like Mathematica and Excel have been using it for years.
- ▶ The value NaN indicates that the result of an operation can't be defined: it's not a regular number, not zero, and not Infinity.
- ▶ NaN is more of a mathematical concept, rather than a value you can use in your calculations.
- ▶ The `Log()` function, for example, calculates the logarithm of positive values. By default, you can't calculate the logarithm of a negative value. If the argument you pass to the `Log()` function is a negative value, the function will return the value NaN to indicate that the calculations produced an invalid result.



- 
- 
- If you execute these statements, the result will be a NaN. Any calculations that involve the result variable (a NaN value) will yield NaN as a result. The statements will all yield NaN:
  - `result = result + result`
  - `result = 10 / result`
  - `result = result + 1E299`
  - `MsgBox(result)`



# Testing for Infinity and NaN


- To find out whether the result of an operation is a NaN or Infinity, use the `IsNaN` and `IsInfinity` methods of the `Single` and `Double` data type.
  - The `Integer` data type doesn't support these methods, even though it's possible to generate `Infinity` and `NaN` results with `Integers`. If the `IsInfinity` method returns `True`, you can further examine the sign of the `Infinity` value with the `IsNegativeInfinity` and `IsPositiveInfinity` methods.
  - In most situations, you'll display a warning and terminate the calculations.
- 

# Handling NaN and Infinity Values

```
Dim var1, var2 As Double
Dim result As Double
var1 = 0
var2 = 0
result = var1 / var2
If result.IsInfinity(result) Then
    If result.IsPositiveInfinity(result) Then
        MsgBox("Encountered a very large number. Can't continue")
    Else
        MsgBox("Encountered a very small number. Can't continue")
    End If
Else
    If result.IsNaN(result) Then
        MsgBox("Unexpected error in calculations")
    Else
        MsgBox("The result is " & result.ToString)
    End If
End If
```



# User-Defined Data Types

- In the previous sections, we assumed that applications create variables to store individual values. As a matter of fact, most programs store sets of data of different types.
  - For example, a program for balancing your checkbook must store several pieces of information for each check: the check's number, amount, date, and so on. All these pieces of information are necessary to process the checks, and ideally, they should be stored together.
- 



# Define a record in VB.NET

```
Structure structureName
    Dim variable1 As varType
    Dim variable2 As varType
    ...
    Dim variablen As varType
End Structure
```



# The Nothing Value

- ▶ The Nothing value is used with Object variables and indicates a variable that has not been initialized. If you want to disassociate an Object variable from the object it represents, set it to Nothing.
- ▶ The following statements create an Object variable that references a Brush, use it, and then release it:
  - ▶ `Dim brush As System.Drawing.Brush`
  - ▶ `brush = New System.Drawing.Brush(bmap)`
  - ▶ `{ use brush object to draw with }`
  - ▶ `brush = Nothing`





# Examining Variable Types

- Besides setting the types of variables and the functions for converting between types, Visual Basic provides two methods that let you examine the type of a variable.
- They are the GetType() and GetTypeCode() methods.
- The GetType() method returns a string with the variable's type ("Int32", "Decimal", and so on).
- The GetTypeCode() method returns a value that identifies the variable's type.



GETTYPE()	GETTYPECODE()	DESCRIPTION
Boolean	3	Boolean value
Byte	6	Byte value (0 to 255)
Char	4	Character
DateTime	16	Date/time value
Decimal	15	Decimal
Double	14	Double-precision floating-point number
Int16	7	2-byte integer (Short)
Int32	9	4-byte integer (Integer)
Int64	11	8-byte integer (Long)
Object		Object (a non-value variable)
SByte	5	Signed byte (-127 to 128)
Single	13	Single-precision floating-point number
String	8	String
UInt16	8	2-byte unsigned integer
UInt32	10	4-byte unsigned integer
UInt64	12	8-byte unsigned integer






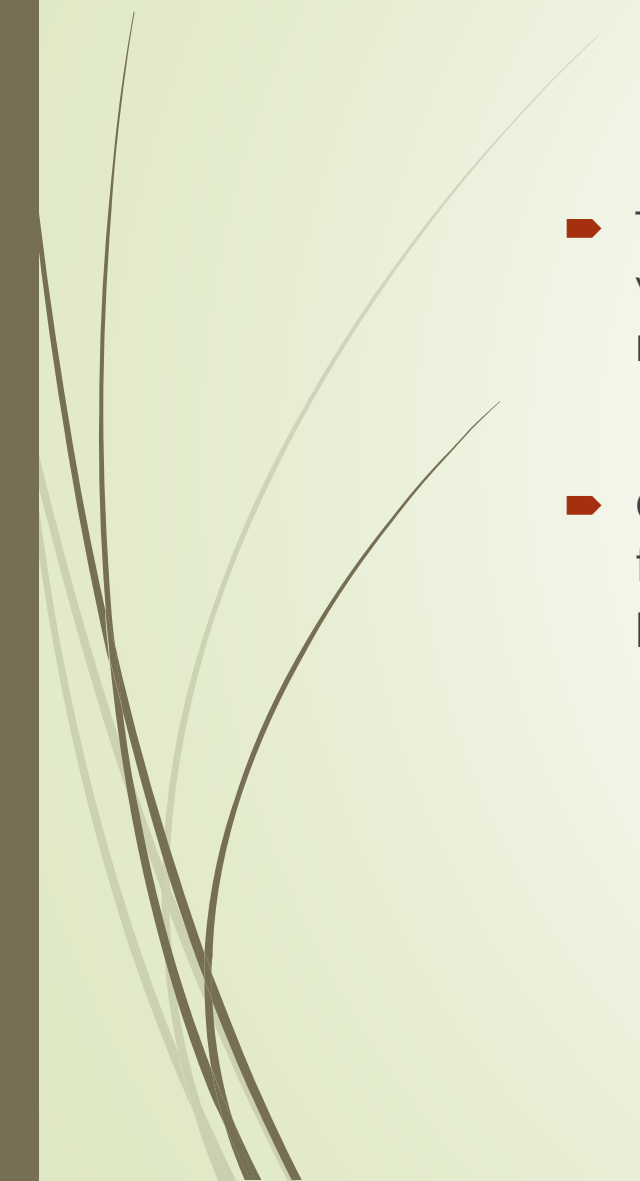
# A Number or a String

- Another set of Visual Basic functions returns variables' data types, but not the exact type. They return a broader type, such as “numeric” for all numeric data types.
- This is the type you usually need in your code. The following functions are used to validate user input, as well as data stored in files, before you process them.
- IsNumeric()
  - Returns True if its argument is a number
- IsArray()
- IsDate()



# Constants


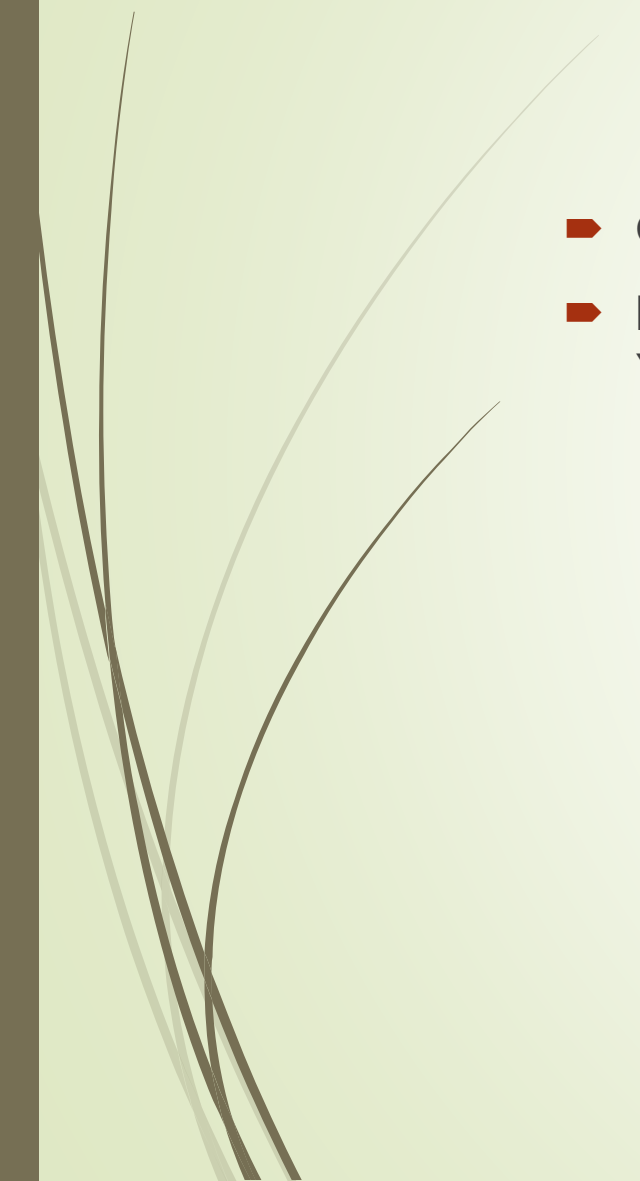
- Some variables don't change value during the execution of a program. These are constants that appear many times in your code.
- For instance, if your program does math calculations, the value of pi (3.14159...) may appear many times. Instead of typing the value 3.14159 over and over again, you can define a constant, name it pi, and use the name of the constant in your code. The statement
  - `circumference = 2 * pi * radius`
- is much easier to understand than the equivalent
  - `circumference = 2 * 3.14159 * radius.`

- 
- 
- You could declare pi as a variable, but constants are preferred for two reasons:
    - Constants don't change value.
    - This is a safety feature. Once a constant has been declared, you can't change its value in subsequent statements, so you can be sure that the value specified in the constant's declaration will take effect in the entire program.
    - Constants are processed faster than variables.
    - When the program is running, the values of constants don't have to be looked up. The compiler substitutes constant names with their values, and the program executes faster.
  - a

- 
- 
- The manner in which you declare constants is similar to the manner in which you declare variables, except that in addition to supplying the constant's name, you must also supply a value, as follows:
    - `Const constantname As type = value`
  - Constants also have a scope and can be Public or Private. The constant pi, for instance, is usually declared in a module as Public so that every procedure can access it:
    - `Public Const pi As Double = 3.14159265358979`



- 
- 
- The best way to define the value of the pi variable is to use the pi member of the Math class:
    - `pi = Math.pi`
  - However, you can't use this assignment in the constant declaration. You must supply the actual value.
  - Constants can be strings, too, like these:
    - `Const ExpDate = #31/12/1997#`
    - `Const ValidKey = "A567dfe"`

- 
- 
- Constant declarations may include other constants.
  - In math calculations, the value  $2 \times \pi$  is almost as common as the value  $\pi$ . You can declare these two values as constants:
    - `Public Const pi As Double = 3.14159265358979`
    - `Public Const pi2 As Double = 2 * pi`



# Arrays



- Unlike simple variables, arrays must be declared with the Dim (or Public, or Private) statement followed by the name of the array and the index of the last element in the array in parentheses—for example,
  - Dim Salaries(15) As Integer
- Salaries is the name of an array that holds 16 values (the salaries of the 16 employees), with indices ranging from 0 to 15.
- Salaries(0) is the first person's salary, Salaries(1) the second person's salary, and so on.



# Declaring Arrays

- All you have to do is remember who corresponds to each salary, but even this data can be handled by another array. To do this, you'd declare another array of 16 elements as follows:
  - `Dim Names(15) As String`
- and then assign values to the elements of both arrays:
  - `Names(0) = "Joe Doe"`
  - `Salaries(0) = 34000`
  - `Names(1) = "Beth York"`
  - `Salaries(1) = 62000`
  - ...
  - `Names(15) = "Peter Smack"`
  - `Salaries(15) = 10300`

# Initializing Arrays

- ▶ Just as you can initialize variables in the same line where you declare them, you can initialize arrays, too, with the following constructor:
  - ▶ `Dim arrayname() As type = {entry0, entry1, ... entryN}`
- ▶ Example that initializes an array of strings:
  - ▶ `Dim names() As String = {"Joe Doe", "Peter Smack"}`
- ▶ This statement is equivalent to the following statements, which declare an array with two elements and then set their values:
  - ▶ `Dim names(1) As String`
  - ▶ `names(0) = "Joe Doe"`
  - ▶ `names(1) = "Peter Smack"`



# Array Limits

- ▶ The first element of an array has index 0. The number that appears in parentheses in the Dim statement is one less than the array's total capacity and is the array's upper limit (or upper bound).
- ▶ The index of the last element of an array (its upper bound) is given by the function UBound(), which accepts as argument the array's name. For the array
  - ▶ Dim myArray(19) As Integer
- ▶ its upper bound is 19, and its capacity is 20 elements.





# Flow-Control Statements



- ▶ Test Structures An application needs a built-in capability to test conditions and take a different course of action depending on the outcome of the test. Visual Basic provides three such decision structures:
  - ▶ If...Then
  - ▶ If...Then...Else
  - ▶ Select Case



# If...Then

- The If...Then statement tests the condition specified; if it's True, the program executes the statement(s) that follow. The If structure can have a single-line or a multiple-line syntax.
- To execute one statement conditionally, use the single-line syntax as follows:
  - If <condition> Then <statement>
- Visual Basic evaluates the condition, and if it's True, executes the statement that follows. If the condition is False, the application continues with the statement following the If statement. You can also execute multiple statements by separating them with colons:
  - If <condition> Then <statement>: <statement>: <statement>




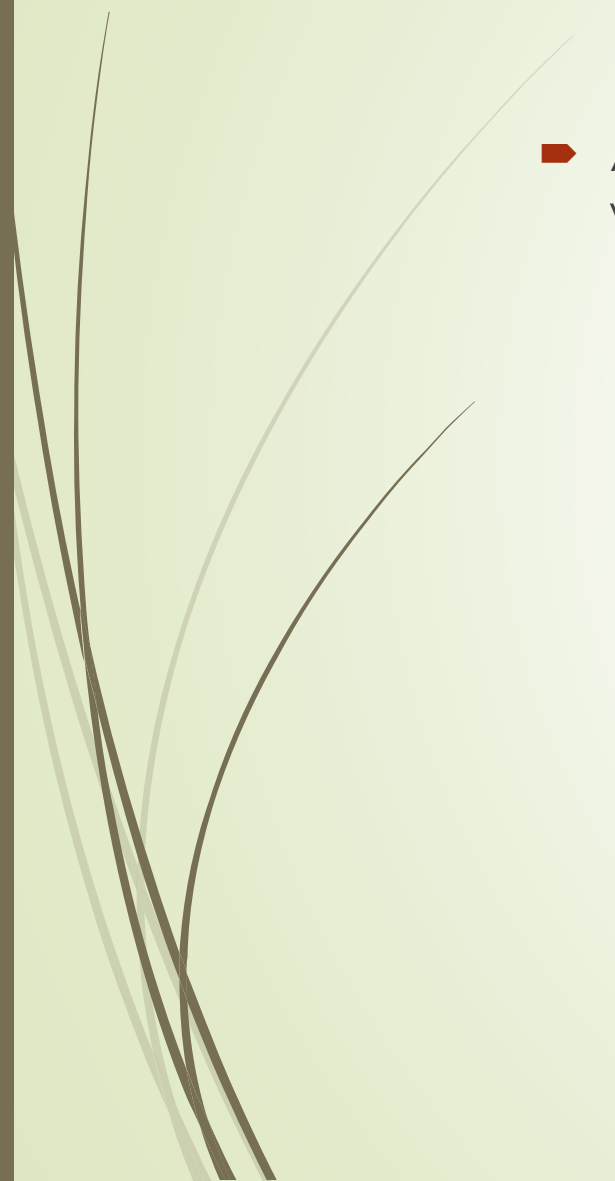
# Example

- Here's an example of a single-line If statement
  - If Month(expDate) > 12 Then expYear = expYear + 1: expMonth = 1
- You can break this statement into multiple lines by using End If, as shown here:
  - If expDate.Month > 12 Then
    - expYear = expYear + 1
    - expMonth = 1
  - End If




# If...Then...Else

- A variation of the If...Then statement is the If...Then...Else statement, which executes one block of statements if the condition is True and another block of statements if the condition is False.
- Visual Basic evaluates the condition; if it's True, VB executes the first block of statements and then jumps to the statement following the End If statement. If the condition is False, Visual Basic ignores the first block of statements and executes the block following the Else keyword.
- The syntax of the If...Then...Else statement is as follows:
  - If <condition> Then
    - <statementblock1>
  - Else
    - <statementblock2>
  - End If


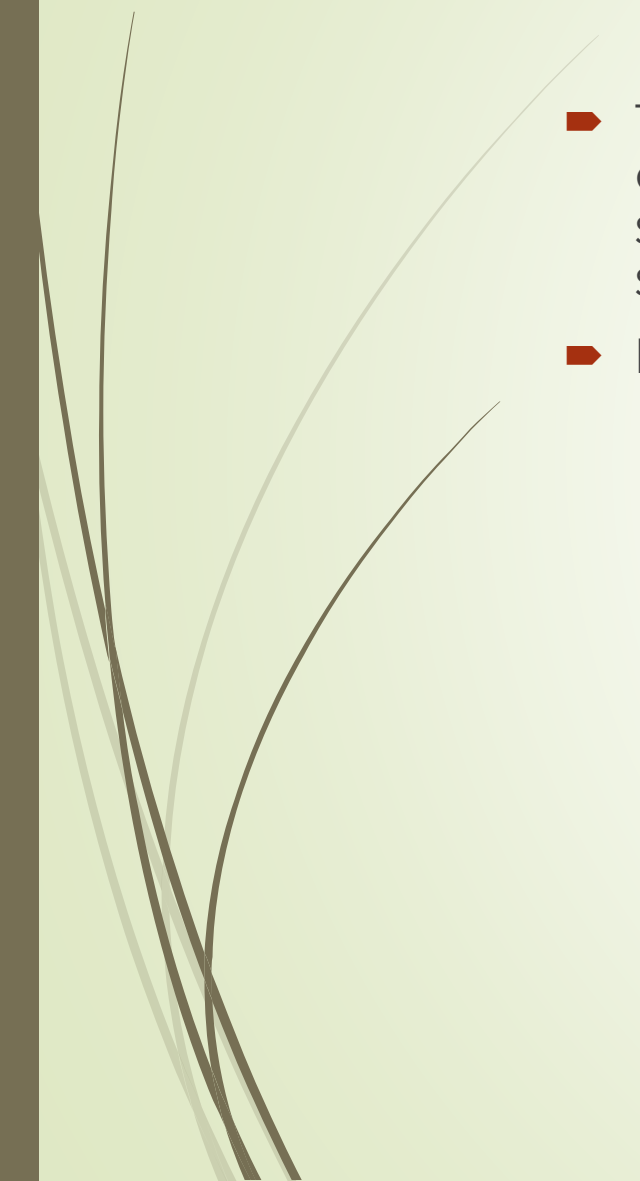
- 
- 
- Another variation of the If...Then...Else statement uses several conditions, with the Elself keyword:
    - If <condition1> Then
      - <statementblock1>
    - Elself <condition2> Then
      - <statementblock2>
    - Elself <condition3> Then
      - <statementblock3>
    - Else
      - <statementblock4>
    - End If



# Select Case

- An alternative to the efficient, but difficult-to-read, code of the multiple-Elseif structure is the Select Case structure, which compares one expression to different values.
  - The advantage of the Select Case statement over multiple If...Then...Else/Elseif statements is that it makes the code easier to read and maintain.
- 



- 
- 
- The Select Case structure tests a single expression, which is evaluated once at the top of the structure. The result of the test is then compared with several values, and if it matches one of them, the corresponding block of statements is executed.
  - Here's the syntax of the Select Case statement:
    - Select Case <expression>
      - Case value1
        - statementblock1
      - Case value2
        - statementblock2 .
        - .
        - .
      - Case Else
        - statementblockN
    - End Select



# Exercises: Using the Select Case Statement

```
Dim message As String
Select Case Now.DayOfWeek
    Case DayOfWeek.Monday
        message = "Have a nice week"
    Case DayOfWeek.Friday
        message = "Have a nice weekend"
    Case Else
        message = "Welcome back!"
End Select
MsgBox(message)
```



# Loop Structures


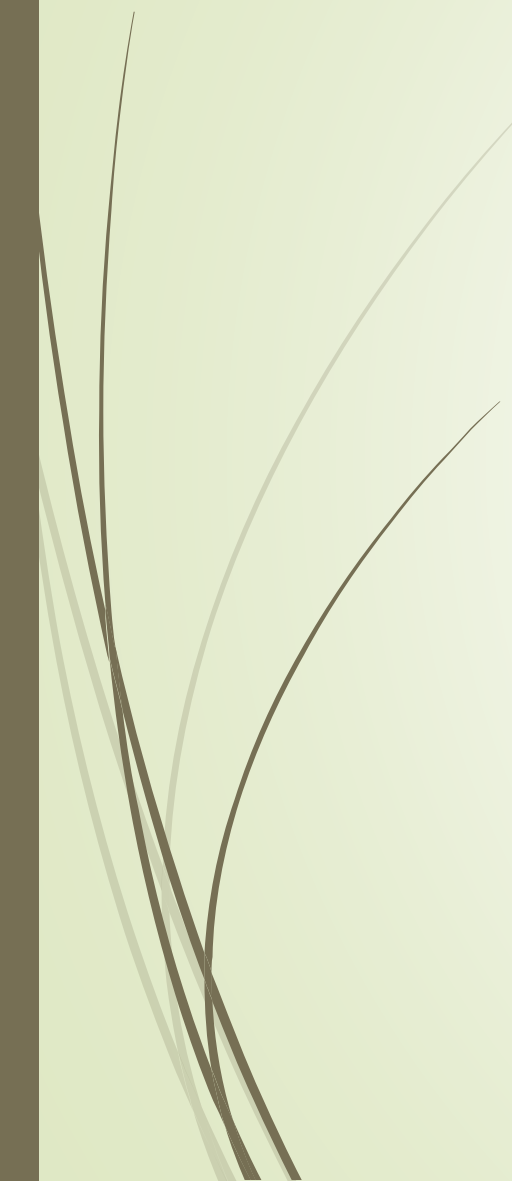



- ▶ Loop structures allow you to execute one or more lines of code repetitively. Many tasks consist of trivial operations that must be repeated over and over again, and looping structures are an important part of any programming language. Visual Basic supports the following loop structures:
  - ▶ For...Next
  - ▶ Do...Loop
  - ▶ While...End While



# For...Next

- The For...Next loop is one of the oldest loop structures in programming languages. Unlike the other two loops, the For...Next loop requires that you know how many times the statements in the loop will be executed. The For...Next loop uses a variable (it's called the loop's counter) that increases or decreases in value during each repetition of the loop. The For...Next loop has the following syntax:
  - For counter = start To end [Step increment]
    - statements
  - Next [counter]
- The keywords in the square brackets are optional.
- The arguments counter, start, end, and increment are all numeric.
- The loop is executed as many times as required for the counter to reach (or exceed) the end value.

- 
- 
- In executing a For...Next loop, Visual Basic completes the following steps:
    - 1. Sets counter equal to start
    - 2. Tests to see if counter is greater than end. If so, it exits the loop. If increment is negative, Visual Basic tests to see if counter is less than end. If it is, it exits the loop.
    - 3. Executes the statements in the block
    - 4. Increments counter by the amount specified with the increment argument. If the increment argument isn't specified, counter is incremented by 1.
    - 5. Repeats the statements







# Exercise: Iterating an Array with a For...Next Loop

- This For...Next scans all the elements of the numeric array data and calculates their average.

```
Dim i As Integer, total As Double
For i = 0 To data.GetUpperBound(0)
    total = total + data(i)
Next i
Console.WriteLine (total / data.Length)
```




- 
- 
- The single most important thing to keep in mind when working with For...Next loops is that the loop's counter is set at the beginning of the loop. Changing the value of the end variable in the loop's body won't have any effect.
  - For example, the following loop will be executed 10 times, not 100 times:
    - `endValue = 10`
    - `For i = 0 To endValue`
      - `endValue = 100`
      - `{ more statements }`
    - `Next i`

- 
- 
- You can, however, adjust the value of the counter from within the loop. The following is an example of an endless (or infinite) loop:
    - For i = 0 To 10
      - Console.WriteLine(i)
      - i = i - 1
    - Next i
  - \*Manipulating the counter of a For...Next loop is strongly discouraged. This practice will most likely lead to bugs such as infinite loops, overflows, and so on. If the number of repetitions of a loop isn't known in advance, use a Do...Loop or a While...End While structure



# Do...Loop

- The Do...Loop executes a block of statements for as long as a condition is True. Visual Basic evaluates an expression, and if it's True, the statements are executed.
  - When the end of block is reached, the expression is evaluated again and, if it's True, the statements are repeated. If the expression is False, the program continues and the statement following the loop is executed.
- 



# Two Variations

- There are two variations of the Do...Loop statement; both use the same basic model. A loop can be executed either while the condition is True or until the condition becomes True. These two variations use the keywords While and Until to specify how long the statements are executed.
- To execute a block of statements while a condition is True, use the following syntax:
  - Do While condition
    - statement-block
  - Loop
- To execute a block of statements until the condition becomes True, use the following syntax:
  - Do Until condition
    - statement-block
  - Loop



# While...End While

- ▶ The While...End While loop executes a block of statements as long as a condition is True. The While loop has the following syntax:
  - ▶ While condition
    - ▶ statement-block
  - ▶ End While
- ▶ If condition is True, all statements are executed and, when the End While statement is reached, control is returned to the While statement, which evaluates condition again.
- ▶ If condition is still True, the process is repeated. If condition is False, the program resumes with the statement following End While.

# Exercise: Reading an Unknown Number of Values

- ▶ This loop prompts the user for numeric data. The user can type a negative value to indicate that all values are entered.

```
Dim number, total As Double
number = 0
While number >= 0
    total = total + number
    number = InputBox("Please enter another value")
End While
```





# Exercise

- Recognize the following types of variables:
- True/False
- 10
- "There are approximately 29,000 words in this chapter"
- #8/27/2001 6:29:11 PM#



# Exercise

- For the Array:
  - Dim testArray(9) As Integer
- What is its upper bound? And what is its capacity?