**Unit -1**

**Introduction to Visual Basic 6**

1.1 The concept of computer programming

Before we begin learning Visual Basic 6 programming, let us understand some basic concepts of programming. Programming means designing a set of instructions to ask the computer to carry out certain jobs that are very much faster and more efficient than human beings can do. As the mirochips of  a CPU can only understand 0 and 1 in the binary system, the earliest programming language uses combinations of 0 and 1 code to communicate with computer, a language that is called machine language. Machine language is extremely difficult to learn . Fortunately , scientists have invented human language-like program languages or high level programming languges which are much easier to master. Some of the high level programming languages are Fortran, Cobol, Java, C, C++, c#, Visual Basic,Turbo Pascal, flash action script, JavaScript,HTML and more.

1.2 What is Visual Basic?

VISUAL BASIC is a high level programming language which  evolved from the earlier DOS version called BASIC. BASIC means Beginners' All-purpose Symbolic Instruction Code. It is a relatively easy programming language to learn. The code looks a lot  like English Language. Different software companies produced different versions of BASIC, such as Microsoft QBASIC, QUICKBASIC, GWBASIC ,IBM BASICA and so on. However, people prefer to use Microsoft Visual Basic today, as it is a well developed programming language and supporting resources are available everywhere. Now, there are many versions of VB exist in the market, the most popular one and still widely used by many VB programmers is none other than [Visual Basic 6](http://www.amazon.com/gp/product/141962895X/ref=as_li_qf_sp_asin_tl?ie=UTF8&tag=liewvoonkiong&linkCode=as2&camp=1789&creative=9325&creativeASIN=141962895X)http://www.assoc-amazon.com/e/ir?t=liewvoonkiong&l=as2&o=1&a=141962895X . We also have VB.net, Visual Basic 2005, [Visual Basic 2008](http://www.vbtutor.net/vb2008/vb2008tutor.html) , [Visual Basic 2010](http://www.vbtutor.net/vb2010/index.html), [Visual Basic 2012](http://www.vbtutor.net/index.php/visual-basic-2012-tutorial/) and [Visual Basic 2013](http://www.vbtutor.net/index.php/visual-basic-2013-tutorial/) . VB2008, VB2010, VB2012 and VB2013 are fully object oriented programming (OOP) languages.

VISUAL BASIC is also a VISUAL and  Event-driven Programming Language. These are the main divergence from the old BASIC. In BASIC, programming is done in a text-only environment and the program is executed sequentially. In VB6, programming is done in a graphical environment. In the old BASIC, you have to write program code for each graphical object you wish to display it on screen, including its position and its color. However, In VB6 , you just need to drag and drop any graphical object anywhere on the form, and you can change its properties  using the properties window.

In addition, Visual Basic 6 is Event-driven because we need to write code in order to perform some tasks in response to certain events. The events usually comprises but not limited to the user's inputs. Some of the events are load, click, double click, drag and drop, pressing the keys and more. We will learn more about events in later lessons. Therefore, a VB6 Program is made up of many subprograms, each has its own program code, and each can be executed independently and at the same time each can be linked together in one way or another.

1.3 What programs can you create with Visual Basic 6?

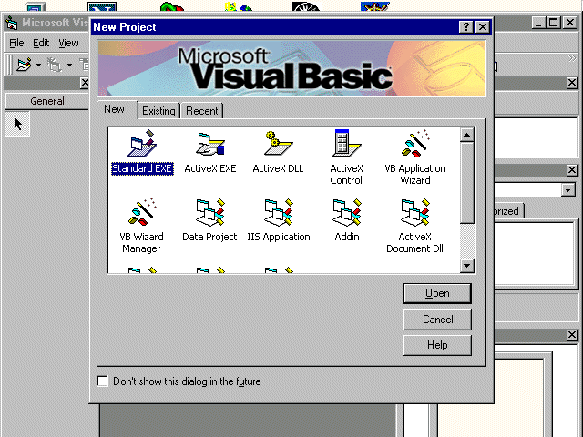
With VB 6, you can create any program depending on your objective.For math teachers, you can create mathematical programs such as [Geometric Progression](http://www.vbtutor.net/VB_Sample/GP.htm), [Quadratic Equation Solver](http://www.vbtutor.net/VB_Sample/QESolver.htm), [Simultaneous Equation Solver](http://www.vbtutor.net/VB_Sample/simuleq.htm) ,[Prime Number](http://www.vbtutor.net/VB_Sample/Prime.htm), [Factors Finder](http://www.vbtutor.net/VB_Sample/factors%20Finders.html), [Quadratic Function Graph Plotter](http://www.vbtutor.net/VB_Sample/QGraphplotter.htm) and so on. For science teachers, you can create simulation programs such as [Projectile](http://www.vbtutor.net/VB_Sample/projectile.htm), [Simple Harmonic Motion](http://www.vbtutor.net/VB_Sample/shm.htm), [Star War](http://www.vbtutor.net/VB_Sample/starwar.htm)  etc. If you are in business, you can also create [business](http://onlinebizguide4you.com/) applications such as [inventory management system](http://www.vbtutor.net/index.php/2013/02/03/inventory-management-system-2/) , [Amortization Calculator](http://www.vbtutor.net/VB_Sample/amortize.htm) , [investments calculator](http://www.vbtutor.net/VB_Sample/Investment.htm), point-of-sale system, payroll system, accounting program and more to help manage your business and increase productivity. For those of you who like games , you can create those programs such as [slot machine](http://www.vbtutor.net/VB_Sample/vbslot2.htm), [reversi](http://www.vbtutor.net/VB_Sample/reversi.htm), [tic tac toe](http://www.vbtutor.net/VB_Sample/tictactoe.htm) and more. Besides, you can create multimedia programs such as [Smart Audio Player](http://www.vbtutor.net/VB_Sample/audio.html), [Multimedia Player](http://www.vbtutor.net/VB_Sample/multimp.htm)  and more. Indeed, there is no limit to what program you can create ! We offer many [sample codes](http://www.vbtutor.net/VB_Sample/sample.html) in our tutorial.

1.4 The Visual Basic 6 Integrated Development Environment

Before you can program in VB 6, you need to install Visual Basic 6 compiler in your computer. You can purchase a copy of [Microsoft Visual Basic 6.0 Learning Edition](http://www.amazon.com/gp/product/B00002SFK8/ref=as_li_qf_sp_asin_tl?ie=UTF8&tag=liewvoonkiong&linkCode=as2&camp=1789&creative=9325&creativeASIN=B00002SFK8)http://www.assoc-amazon.com/e/ir?t=liewvoonkiong&l=as2&o=1&a=B00002SFK8 or [Microsoft Visual Basic Professional 6.0 with Plus Pack](http://www.amazon.com/gp/product/B00002S7HR/ref=as_li_qf_sp_asin_tl?ie=UTF8&tag=liewvoonkiong&linkCode=as2&camp=1789&creative=9325&creativeASIN=B00002S7HR)http://www.assoc-amazon.com/e/ir?t=liewvoonkiong&l=as2&o=1&a=B00002S7HR from Amazon.com, both are vb6 compilers. If you have already installed Microsoft Office in your PC or laptop, you can also use the built-in [Visual Basic Application in Excel](http://excelvbatutor.com/) to start creating Visual Basic programs without having to spend extra cash to buy the VB6 compiler.

After installing vb6 compiler, the icon with appear on your desktop or in your programs menu. Click on the icon to launch the VB6 compiler. On start up, Visual Basic 6.0  will display the following dialog box as shown in Figure 1.1.

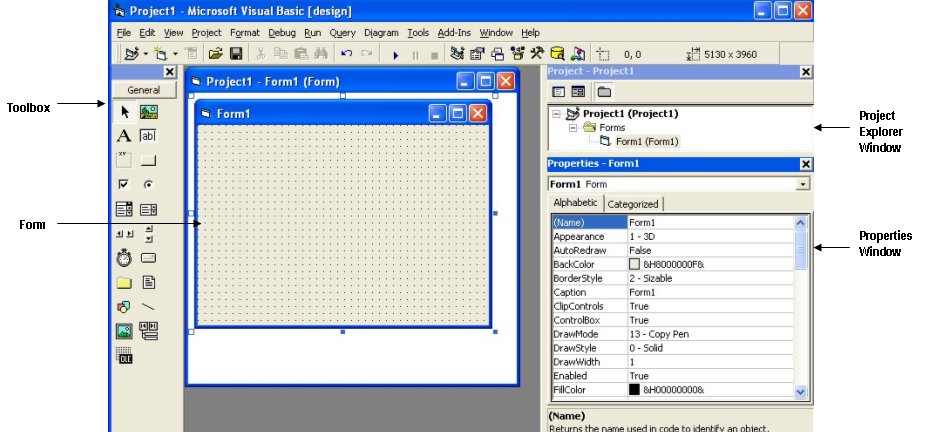
Figure 1.1: New Project Dialog



You can choose to either start a new project, open an existing project or select a list of recently opened programs. A project is a collection of files that make up your application. There are various types of applications that we could create, however, we shall concentrate on creating Standard EXE programs (EXE means executable). Before you begin, you must think of an application that might be useful, have commercial values . educational or recreational. click on the Standard EXE icon to go into the actual Visual Basic 6 programming environment.

When you start a new Visual Basic 6 Standard EXE project, you will be presented  with the Visual Basic 6 Integrated Development Environment (IDE). The Visual Basic 6 Integrated Programming Environment is show in Figure 1.2. It consists of the toolbox, the form, the project explorer and the properties window.

Figure 1.2: VB6 Programming Environment



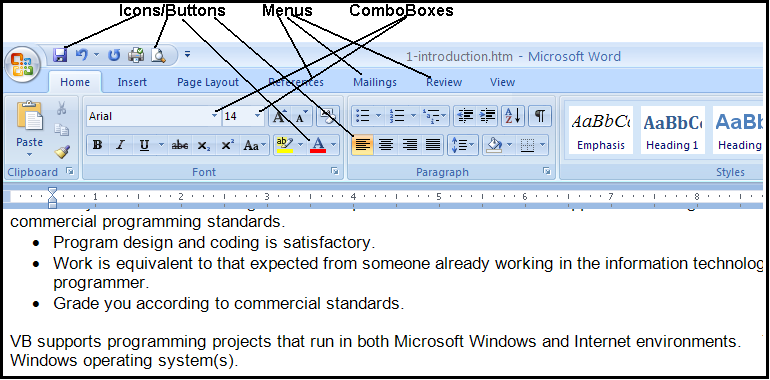
Form is the primary building block of a Visual Basic 6 application. A Visual Basic 6 application can actually comprises many forms; but we shall focus on developing an application with one form first. We will learn how to develop applications with multiple forms later. Before you proceed to build the application, it is a good practice to save the project first. You can save the project by selecting Save Project from the File menu, assign a name to your project and save it in a certain folder.

[Bookmark and Share](http://www.addthis.com/bookmark.php?v=250&pub=xa-4a8273cd12365f70)

## Commercial GUI Design/Development Approach

Modern programs run in a **graphical user interface** (**GUI**) environment. You can open up any desktop program or web browser and examine the **GUI**. This interface includes:

* Buttons (some with Icons and Graphical Images) and Menus on which you point and click.
* TextBox controls in which you type information.
* Graphical images to guide a system user in using the program.
* Other components.
* A GUI for a version of Microsoft Word is shown in this figure.



# Microsoft Visual Studio and Visual Basic (.NET Version)

## Programming Languages

Microsoft's **Visual Studio** (also called **Visual Studio.NET**) includes several different programming languages:

* **Visual Basic**,
* **Visual C#** (C sharp),
* **Visual C++** (cee-plus-plus),
* **Visual F#** (F sharp),
* **JScript** and
* Web Development (called **ASP.NET**).
* It also includes the **.NET 4 Framework** upon which these languages operate.

All of these languages **compile.**

* This means they are translated from **human readable-form** to **machine readable-form** to the same **Microsoft Intermediate Language** (MSIL).
* MSIL run within the **Common Language Runtime** (CLR) – a component of the .NET Framework.

Visual Basic (also termed Visual Basic.NET or VB) is a major revision of earlier Microsoft VB products.

* This latest version is Version 10 of VB.
* This is an upgrade of earlier**.NET** versions.
* The current and earlier versions are not completely compatible—a program coded with an earlier version that is upgraded to the current version cannot be opened again with the earlier version software.
* Version 6 and earlier are not .NET-compatible – programs created with these versions are completely incompatible with .NET applications.

VB is available in several editions including the free **Express Edition** that you can download from Microsoft. Other editions (**Professional**, **Premium**, and **Ultimate**) are used within industry.

* Our classroom has the **Visual Studio 2010 Ultimate Edition** installed.
* You can also use the **Express Edition (2010)** (free download) or,
* The **Professional Edition (2010)** (free download for students).
* You must use one of these 2010 editions (also called Version 10) for this class.

## Visual Studio Components

When Visual Studio is installed on a computer, there are two mandatory components to the installation and an optional third component.

* **.NET Framework Class Library**. This is a library of predefined class objects. It enables you to quickly build a computer application through the use of predefined objects such as forms, text boxes, labels, buttons, drop-down list controls, and others (mandatory).
* **Common Language Runtime (CLR)**. This component manages the execution of a programming project written in any of the languages that are included within Visual Studio including Visual Basic as a language (mandatory). This component is installed as part of the .NET Framework.
* **MSDN (Help)**. This is the help component and provides access to a help reference library. This is covered in detail at the end of this set of notes. It is an optional, but highly recommended component.

## Object-Oriented Programming Terminology

VB is an ***object-oriented programming language***.

* Means you work with **objects** in building an application.
* Examples: Form objects, Button objects, TextBox objects, Label objects, ListBox objects, PictureBox objects, and more.

VB is also termed an ***event-driven programming language*** because you will write program code that responds to events that are controlled by the system user. Example events include:

* Clicking a button or menu.
* Opening or Closing a form.
* Moving the mouse over the top of an object such as a text box.
* Moving from one text box to another.

In order to work with VB, you need to understand "**object**" terminology as defined in Table 1.

|  |  |
| --- | --- |
| **Table 1** | |
| **Terminology** | **Definition** |
| **Object** | A thing – like a noun in English. Examples include **forms** and controls you place on forms such as **buttons**, **text boxes**, and **icons**. |
| **Property** | Objects have properties – like adjectives in English. Properties describe object behaviors. Examples of properties include **Text**, **Name**, **BackColor**, **Font**, and **Size**.  Refer to a property by the notation **ObjectName.PropertyName** (use the .dot notation) – example: **TotalDueTextBox.Text** or **AccountLabel.ForeColor**. |
| **Method** | Like a verb in English – these are the actions that objects exhibit. Examples include methods to **Show** and **Hide** forms and methods to **Print** and **Close** forms.  Refer to a method with the notation **ObjectName.MethodName** – example **Me.Close** will close the current form. |
| **Event** | Events are actions usually triggered by the system user such as clicking a button; however, events can also be triggered by the actions of objects. For example, closing a form can trigger an event. |
| **Class** | This is a really abstract term – it is a sort of **template** for an object. For example, all forms belong to the **Form** class of object. All buttons belong to the **Button** class of object. Classes include definitions for object properties, methods, and associated events. Each class is assigned an identifying **namespace** within the .NET Framework Class Library.  Each new object you create is defined based on its class – the new object is called a **class instance**. |

# Getting Started with Visual Studio

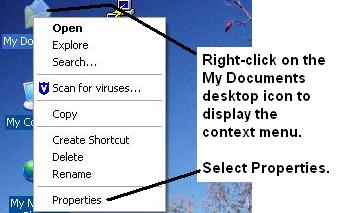
You will use the VB component of Visual Studio to create and test projects.

* The programming applications you will design and develop are called ***solutions*** in VB.
* A ***solution*** can actually contain more than one ***project***, but in this course we will focus on just creating a single project in a solution.
* Each solution is stored in a folder identified by the solution name.
* This is covered in more detail later in this set of notes.

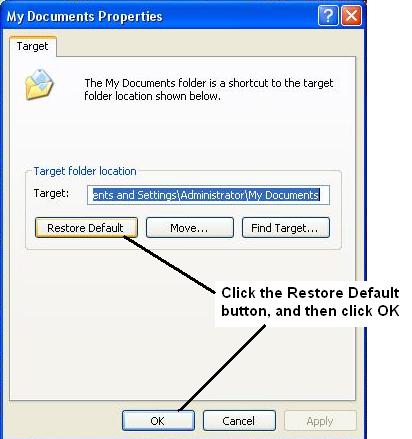
## Before Using Visual Basic

When you first startup a computer in our classroom and computer laboratories, you may need to reset the **Properties** of **My** **Documents**.

* Required because the operating system setup on these classroom/lab computers is produced from a “ghost image” created by the system administrator.
* At the desktop, right-click on the **My Documents** icon.
* Select the **Properties** menu option as shown in this figure.

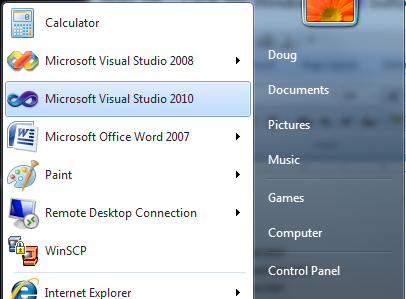


* Click the **Restore Defaults button** and then **OK** as shown in the below figure – this sets the ownership to you as the logged in user of the computer.
* Do this each time you plan to use VB in the classroom or computer lab – this is NOT necessary when using VB on your home computer.



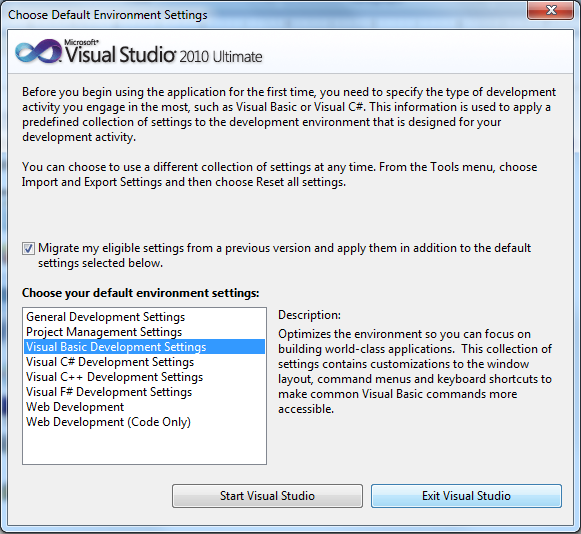
## Launching Visual Basic

Start VB – Click the Windows **Start** button and locate **Microsoft Visual Studio 2010** – launch the program by clicking. The option will look something like this.



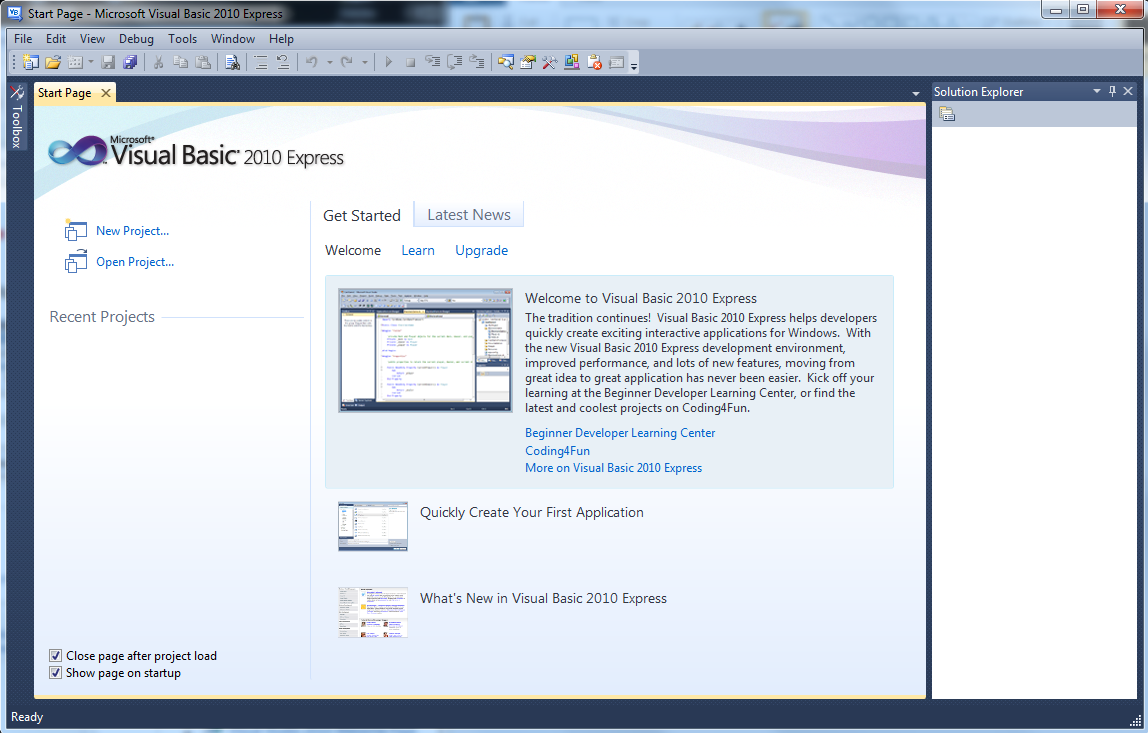
The first screen to display may require you to choose your **Default Environment Settings**.

* If this screen displays, click the **Visual Basic Development Settings** option in the list box and then click the **Start Visual Studio** button

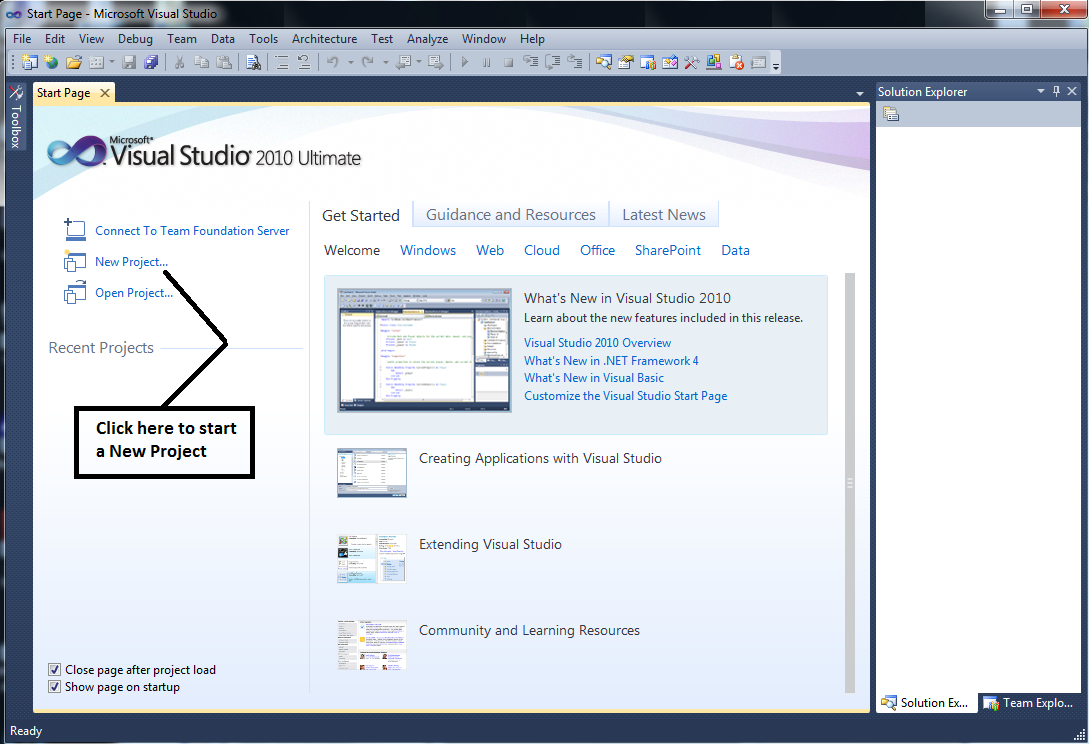


The Visual Studio **Start Page** is shown in the figure below. The start page is slightly different for Visual Studio 2010 (Professional or Ultimate Edition) and Visual Basic 2010 Express Edition.

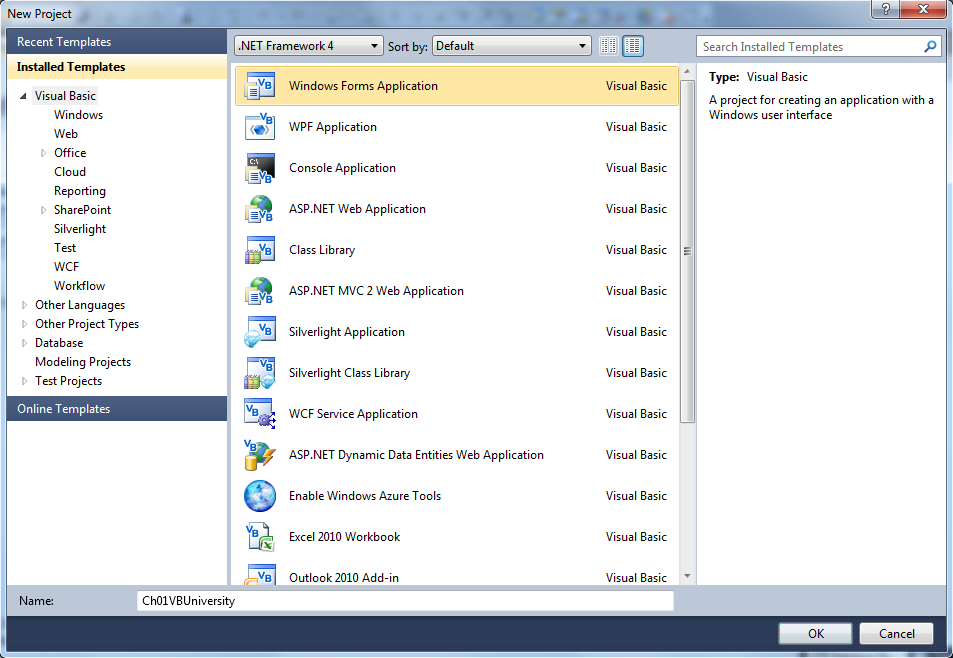
* This is the Start Page for **Visual Basic 2010 Express Edition**.



* This is the Start Page for **Visual Studio 2010** (Ultimate Edition).



Click the **New Project** link shown in the figure above to create a new project. This opens the **New Project** dialog box shown below.

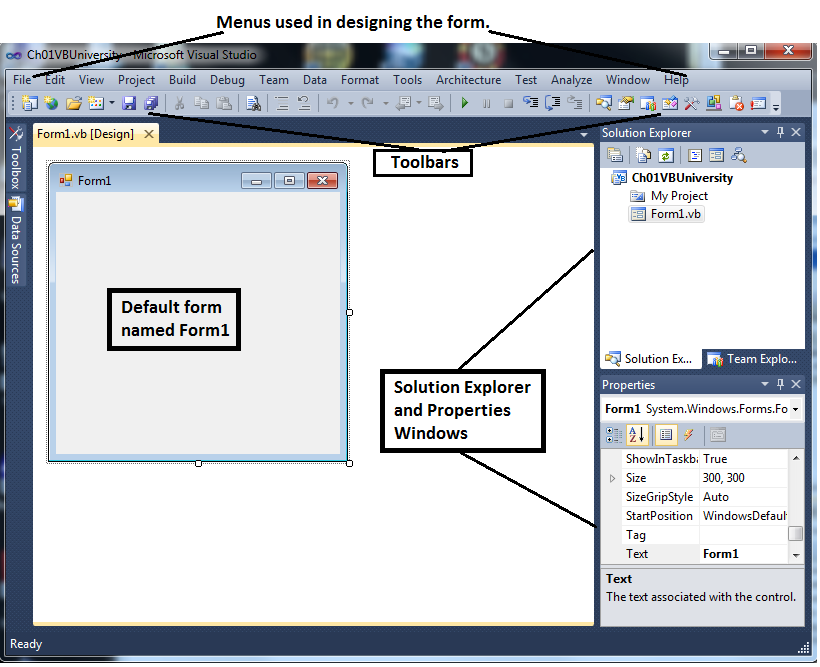


* Your first project will be a **Visual Basic Project** using a **Windows Forms Application** template as shown in the figures above.
* The default name is **WindowsApplication1** -- not a very useful name. Change the project name to: **Ch01VBUniversity**.
* Click the **OK** button – it takes several seconds to create the project files from the template.

## The Integrated Development Environment

Familiarize yourself with the **Integrated Development Environment** (**IDE**).

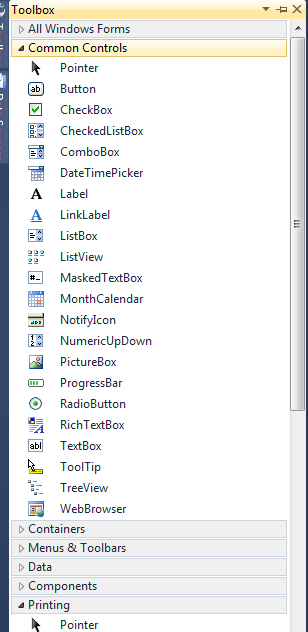
* Across the top are menus with different options used in the designing of an application.
* **Toolbars** with shortcut icons are shown below the menus.
* **Form Designer** (also termed the **Document Window**).
* Displays open documents such as the **Form1** shown in the figure below.
* Tabs along the top are used to allow you to switch rapidly between open documents.
* Every project starts with a default form named **Form1** – the form you modify to build your business application.
* The form can be resized by using the sizing handles.
* The form can be renamed (you’ll do this later in an exercise).
* **Solution Explorer Window** – displays filenames for files that comprise a project. The solution name is also shown here (**Ch01VBUniversity**).
* **Properties Window** – displays properties of the currently selected object – in the figure the properties displayed are those of the Form object.
* **ToolBox Window** – this is shown along the left edge of the figure in collapsed display.



## The Toolbox

The **Toolbox**:

* Contains controls that are used to build forms.
* Can be expanded or collapsed with the **Pin/Unpin** icon.
* Parts of the Toolbox can be expanded/collapsed with the **white and black pointer** icons to the left of the groupings.
* Used to add controls (tools) to a form by either double-clicking or dragging/dropping (your option on which to use).



## Set Up Your Work Environment

The IDE shows the default work environment in terms of the windows that are displayed. Experiment with pinning and unpinning these windows by using the **AutoHide** pushpin icon for each window.

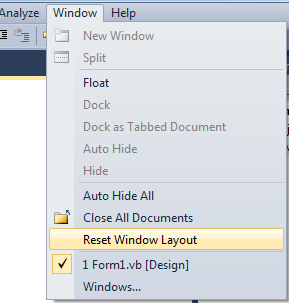
* Toolbox
* Solution Explorer
* Properties

You can also close/open these windows with the **View** menu.

## Resetting the Default IDE Layout

Sometimes you startup VB and the IDE layout is not what you expect.

* Windows you want such as Solution Explorer, Toolbox, or Properties may be closed.
* The quick way to reset the layout is through the **Window** menu, **Reset Window Layout** submenu option as shown in this figure.



### Visual Basic Development Environment

[The Visual Basic Environment (General)](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/VBEnvGeneral.htm)

In the previous topic on [Visual Basic for Applications (VBA) Specifics](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/VBASpecifics.htm), we focused primarily on how to write code in the Visual Basic for Applications (VBA) development environment embedded within ArcMap and ArcCatalog. Now this topic focuses on particular issues related to creating ActiveX DLLs that can be added to the applications and writing external standalone applications using the Visual Basic development environment. More details of using Visual Basic are given with the documentation that accompanies ArcObjects Developer Controls.

[Creating COM components](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/vbspecifics.htm#Creating_COM_components)  
[Implementing interfaces](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/vbspecifics.htm#Implementing_interfaces)  
[Setting references to the ESRI object libraries](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/vbspecifics.htm#Setting_references_to_the_ESRI_object_libraries)  
    [To add a reference to an object library](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/vbspecifics.htm#To_add_a_reference_to_an_object_library)  
[Referring to a document](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/vbspecifics.htm#Referring_to_a_document)  
[Getting to an object](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/vbspecifics.htm#Getting_to_an_object)  
[Running ArcMap with a command line argument](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/vbspecifics.htm#Running_ArcMap_with_a_command_line_argument)  
[Debugging Visual Basic Code](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/vbspecifics.htm#Debugging_Visual_Basic_Code)  
    [Running The Code Within an Application](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/vbspecifics.htm#Running_The_Code_Within_an_Application)  
    [Visual Basic Debugger Issues](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/vbspecifics.htm#Visual_Basic_Debugger_Issues)  
    [Alternatives to the Visual Basic Debugger](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/vbspecifics.htm#Alternatives_to_the_Visual_Basic_Debugger)  
        [Visual C++ Debugger](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/vbspecifics.htm#Visual_C++_Debugger)  
        [ATL Wrapper Classes](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/vbspecifics.htm#ATL_Wrapper_Classes)

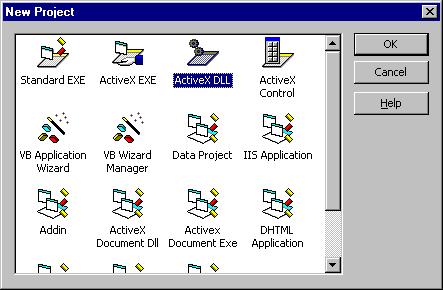
#### Creating COM components

Most developers use Visual Basic to create a COM component that works with ArcMap or ArcCatalog. Earlier in the Getting Started section of the help you learned that since the ESRI applications are COM clients—their architecture supports the use of software components that adhere to the COM specification—you can build components with different languages, including Visual Basic. These components can then be added to the applications easily. For information about packaging and deploying COM components that you've built with Visual Basic, see the [Packaging and Deploying Customizations](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/PackageDeploy.htm) topic.

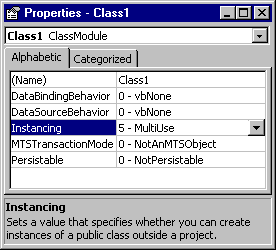
This section is not intended as a Visual Basic tutorial; rather, it highlights aspects of Visual Basic that you should know in order to be effective when working with ArcObjects.

In Visual Basic you can build a COM component that will work with ArcMap or ArcCatalog by creating an ActiveX DLL. This section will review the rudimentary steps involved. Note that these steps are not all-inclusive. Your project may involve other requirements.

1. Start Visual Basic. In the New Project dialog box, create an ActiveX DLL Project.



1. In the Properties window, make sure that the Instancing property for the initial class module and any other class modules you add to the Project is set to 5—MultiUse.



1. Reference the ESRI Object Library.
2. Implement the required interfaces. When you implement an interface in a class module, the class provides its own versions of all the public procedures specified in the type library of the interface. In addition to providing a mapping between the interface prototypes and your procedures, the *Implements* statement causes the class to accept COM *QueryInterface* calls for the specified interface ID. You must include all the public procedures involved. A missing member in an implementation of an interface or class causes an error. If you don't put code in one of the procedures in a class you are implementing, you can raise the appropriate error (*Const E\_NOTIMPL* = &H80004001). That way, if someone else uses the class, they'll understand that a member is not implemented.

The [ESRI Interface Implementer Add-In for VB](http://edndoc.esri.com/arcobjects/8.3/Utilities/InterfaceImpl.htm) can be used to automate steps 3 and 4.

1. Add any additional code that's needed.
2. Establish the Project Name and other properties to identify the component. In the Project Properties dialog box, the Project Name you specify will be used as the name of the component's type library. It can be combined with the name of each class the component provides to produce unique class names (these names are also called ProgIDs). These names appear in the Component Category Manager. Save the project.
3. Compile the DLL.
4. Set the component's Version Compatibility to binary. As your code evolves, it's good practice to set the components to Binary Compatibility so if you make changes to a component, you'll be warned that you're breaking compatibility. For additional information, see the 'Binary compatibility mode' help topic in the Visual Basic online help.

Visual Basic automatically generates the necessary GUIDs for the classes, interfaces, and libraries. Setting binary compatibility forces VB to re-use the GUIDs from a previous compilation of the DLL. This is essential since ArcMap stores the GUIDs of commands in the document for subsequent loading.

1. Save the project.
2. Make the component available to the application. You can add a component to a document or template by clicking the Add from file button in the Customize dialog box's Commands tab. In addition, you can register a component in the Component Category Manager.

#### Implementing interfaces

You implement interfaces differently in Visual Basic depending if they are inbound or outbound interfaces. An outbound interface is seen by Visual Basic as an event source, and is supported through the *WithEvents* keyword. To handle the outbound interface, *IActiveViewEvents*, in Visual Basic (the default outbound interface of the *Map* class), use the *WithEvents* keyword and provide appropriate functions to handle the events.

Private WithEvents ViewEvents As Map

Private Sub ViewEvents\_SelectionChanged()

' User changed feature selection update my feature list form

UpdateM FeatureForm

End Sub

Inbound interfaces are supported with the *Implements* keyword. However, unlike the outbound interface, all the methods defined on the interface must be stubbed out. This ensures that the vTable is correctly formed when the object is instantiated. Not all of the methods have to be fully coded, but the stub functions must be there. If the implementation is blank an appropriate return code should be given to any client to inform them that the method is not implemented (see the section [Working With HRESULTs](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/VBEnvGeneral.htm#Working_With_HRESULTs) in the "The Visual Basic Environment (General)" topic). To implement the *IExtension* interface, code similar to that below is required. Note all the methods are implemented.

Private m\_pApp As IApplication

Implements IExtension

Private Property Get IExtension\_Name() As String

IExtension\_Name = "Sample Extension"

End PropertY

Private Sub IExtension\_Startup(B Ref initializationData As Variant)

Set m\_pApp = initializationData

End Sub

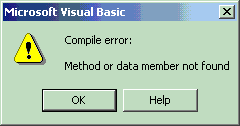
Private Sub IExtension\_Shutdown()

Set m\_pApp = Nothing

End Sub

#### Setting references to the ESRI object libraries

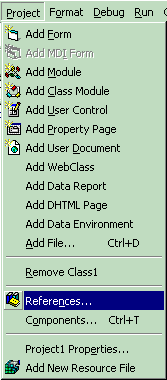
The principle difference between working with the VBA development environment embedded in the applications and working with Visual Basic is that the latter environment requires that you load the appropriate object libraries so that any object variables that you declare can be found. If you don't add the reference, you'll get the error message to below. In addition, the global variables *ThisDocument* and *Application* are not available to you.



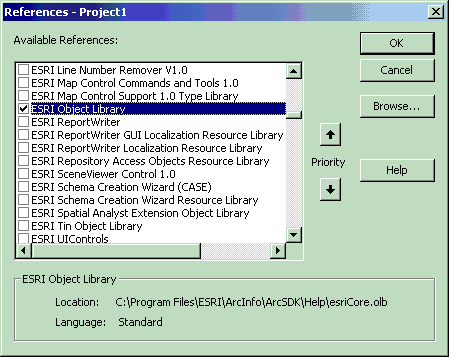
**To add a reference to an object library**

In all cases, you'll need to load the ESRI Object Library esriCore.olb. Depending on what you want your code to do, you may add other ESRI object libraries, perhaps for one of the extensions.

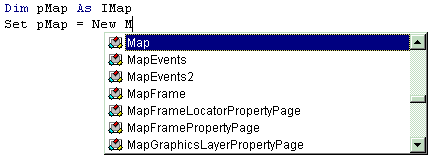
To display the References dialog box in which you can set the references you need, select References in the Visual Basic Project menu.



After you set a reference to an object library by selecting the check box next to its name, you can find a specific object and its methods and properties in the Object Browser.



The References dialog.



After the ESRI Object Library is referenced, all the types contained within it are available to Visual Basic. IntelliSense will also work with the contents of the object library.

If you are not using any objects in a referenced library, you should clear the check box for that reference to minimize the number of object references Visual Basic must resolve, thus reducing the time it takes your project to compile. You should not remove a reference for an item that is used in your project.

You can't remove the "Visual Basic for Applications" and "Visual Basic objects and procedures" references because they are necessary for running Visual Basic.

#### Referring to a document

Each VBA project (Normal, Project, TemplateProject) has a class called *ThisDocument*, which represents the document object. Anywhere you write code in VBA you can reference the document as *ThisDocument*. Further, if you are writing your code in the *ThisDocument* Code window, you have direct access to all the methods and properties on *IDocument*. This is not available in Visual Basic. You must first get a reference to the *Application* and then the document. When adding both extensions and commands to ArcGIS applications, a pointer to the *IApplication* interface is provided. For code samples that show you how to get a handle on the application, see chapter 3, 'Customizing the user interface' in the Exploring ArcObjects book.

Implements IExtension

Private m\_pApp As IApplication

Private Sub IExtension\_Startup(ByRef initializationData As Variant)

Set m\_pApp = initializationData ' Assign IApplication

End Sub

Implements ICommand

Private m\_pApp As IApplication

Private Sub ICommand\_OnCreate(ByVal hook As Object)

Set m\_pApp = hook ' QI for IApplication

End Sub

Now that a reference to the application is in an *IApplication* pointer member variable, the document, and hence all other objects, can be accessed from any method within the class.

Dim pDoc as IDocument

Set pDoc = m\_pApp.Document

MsgBox pDoc.Name

#### Getting to an object

In the previous example, navigating around the objects within ArcMap is a straightforward process since a pointer to the *Application* object, the root object of most of the ArcGIS application's objects, is passed to the object via one of its interfaces. This, however, is not the case with all interfaces that are implemented within the ArcObjects application framework. There are cases when you may implement an object that exists within the framework and there is no possibility to traverse the object hierarchy from that object. This is because very few objects support a reference to their parent object (the *IDocument* interface has a property named *Parent* that references the *IApplication* interface). In order to give developers access to the application object, there is a singleton object that provides a pointer to the running application object. The code below illustrates its use.

Dim pAppRef As New AppRef

Dim pApp as IApplication

Set pApp = pAppRef

Singletons are objects that only support one instance of the object. These objects have a class factory that ensures that anytime an object is requested, a pointer to an already existing object is returned.

You must be careful to ensure that this object is only used where the implementation will only ever run within ArcMap and ArcCatalog. For instance, it would not be a good idea to make use of this function from within a custom feature, since that would restrict what applications could be used to view the feature class.

#### Running ArcMap with a command line argument

You can start ArcMap from the command line and pass it an argument that is either the pathname of a document (.mxd) or the pathname of a template (.mxt). In the former case, ArcMap will open the document; in the latter case ArcMap will create a new document based on the template specified.

You can also pass an argument and create an instance of ArcMap by supplying arguments to the Win32 API's *ShellExecute* function or Visual Basic's *Shell* function, as follows.

Dim ret As Variant

ret = Shell("d:\arcexe83\bin\arcmap.exe \_

d:\arcexe83\bin\templates\LetterPortrait.mxt", vbNormalFocus)

By default, Shell runs other programs asynchronously. This means that ArcMap might not finish executing before the statements following the Shell function are executed.

In Visual Basic, it is not possible to determine the command line used to start the application. There is a sample on disk that provides this functionality. It can be found at \arcgis\arcexe83\ArcObjects Developer Kit\samples\COMTechniques\Command Line.

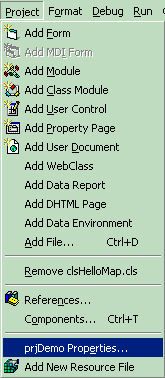
To execute a program and wait until it is terminated, you must call three Win32 API functions. First, call the *CreateProcessA* function to load and execute ArcMap. Next, call the *WaitForSingleObject* function, which forces the operating system to wait until ArcMap has been terminated. Finally, when the user has terminated the application, call the CloseHandle function to release the application's 32-bit identifier to the system pool.

### Debugging Visual Basic Code

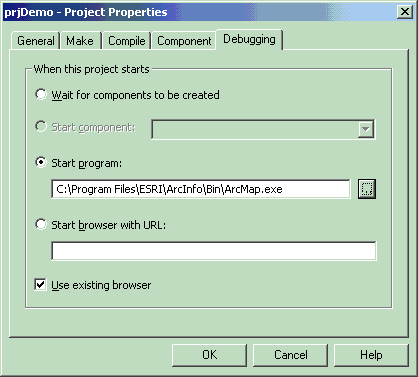
Visual Basic has a debugger integrated into its development environment. This is in many cases a valuable tool when debugging Visual Basic code, however, in some cases it is not possible to use the VB debugger. The use of the debugger and these special cases are discussed below.

#### Running The Code Within an Application

It is possible to use the Visual Basic debugger to debug your ArcObjects-based source code, even when ActiveX DLLs are the target server. The application that will host your DLL must be set as the Debug application. To do this, select the appropriate application, ArcMap.exe, for instance, and set it as the Start Program in the Debugging Options of the Project Properties.



The Projects menu.



The Debugging pane of the Project Properties dialog box.

Using commands on the Debug toolbar, ArcMap can be started and the DLL loaded and debugged. Breakpoints can be set, lines stepped over, functions stepped into and variables checked. Moving the line pointer in the left-hand margin can also set the current execution line.

http://edndoc.esri.com/arcobjects/8.3/gettingstarted/graphics/debugtoolbar.gif

The Debug toolbar.

#### Visual Basic Debugger Issues

In many cases, the Visual Basic debugger will work without any problems, however there are two problems when using the debugger that is supplied with Visual Basic 6. Both of these problems exist because of the way that Visual Basic implements its debugger.

Normally when running a tool within ArcMap, the DLL is loaded into ArcMap's address space and calls are made directly into the DLL. When debugging this is not the case. Visual Basic makes changes to the registry so that the CLSID for your DLL does not point to your DLL, but instead it points to the Visual Basic Debug DLL (VB6debug.dll). The Debug DLL must then support all the interfaces implemented by your class on the fly. With the VB Debug DLL loaded into ArcMap, any method calls that come into the DLL are forwarded on to Visual Basic where the code to be debugged is executed. The two problems with this are caused by the changes made to the Registry and the cross-process space method calling. When these restrictions are first encountered it can be confusing since the object works outside the debugger, or at least until it hits the area of problem code.

Since the method calls made from ArcMap to the custom tool are across apartments, there is a requirement for the interfaces to be marshaled. This marshalling causes problems in certain circumstances. Most data types can be automatically marshaled by the system, but there are a few that require custom code, because the standard marshaler does not support the data types. If one of these data types is used by an interface within the custom tool and there is no custom marshalling code, the debugger will fail with an "Interface not supported error".

The registry manipulation also breaks the support for component categories. Any time there is a request on a component category, the category manager within COM will be unable to find your component, because rather than asking whether your DLL belongs to the component category, COM is asking whether the VB debugger DLL belongs to the component category, which obviously it doesn't. What this means is that anytime a component category is used to automate the loading of a DLL, the DLL cannot be debugged using the Visual Basic debugger.

This obviously causes problems for many of the ways to extend the framework. The most common way to extend the framework is to add a command or tool. Previously it was discussed how component categories were used in this instance. Remember the component category was only used to build the list of commands in the dialog box. This means that if the command to be debugged is already present on a toolbar the Visual Basic debugger can be used. Hence, the procedure for debugging Visual Basic objects that implement the *ICommand* interface is to ensure that the command is added to a toolbar when ArcMap is executed standalone, and then, after saving the document, loading ArcMap through the debugger.

In some cases, like extensions and property pages, it is not possible to use the Visual Basic debugger. If you have access to the Visual C++ Debugger you can use one of the options outlined below. Fortunately, there are a number of ESRI Visual Basic Add-ins that make it possible to track down the problem quickly and effectively. The add-ins described in the topic [ESRI Add-Ins for Visual Basic](http://edndoc.esri.com/arcobjects/8.3/Utilities/ESRIVBAddins.htm), provide error log information, including line and module details. A sample output from an error log is given below; note the call stack information along with line numbers.

Error Log saved on : 8/28/2000 - 0:39:04 AM

Record Call Stack Sequence - Bottom line is error line.

chkVisible\_MouseUp C:\Source\MapControl\Commands\frmLa er.frm Line : 96

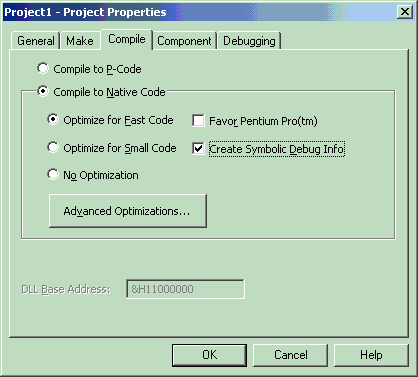
RefreshMap C:\Source\MapControl\Commands\frmLa er.frm Line : 20

Description

Object variable or With block variable not set

#### Alternatives to the Visual Basic Debugger

If the Visual Basic debugger and add-ins do not provide enough information, the Visual C++ debugger can be used, either on its own or with C++ ATL wrapper classes. The Visual C++ debugger does not run the object to be debugged out of process from ArcMap, which means that none of the above issues apply. Common debug commands are given in the section [Debugging Tips In Developer Studio](http://edndoc.esri.com/arcobjects/8.3/gettingstarted/VCppEnv.htm#Debugging_Tips_in_Developer_Studio) in the "The Visual C++ Environment" topic. Both of the techniques below require the Visual Basic project to be compiled with Debug Symbol information.



The Visual C++ Debugger can work with this symbolic debug information and the source files.

**Visual C++ Debugger**

It is possible to use the Visual C++ debugger directly by attaching to a running process that has the Visual Basic object to be debugged loaded, and then setting a breakpoint in the Visual Basic file. When the line of code is reached, the debugger will halt execution and step you into the source file at the correct line. The required steps are shown below.

1. Start an appropriate application, such as ArcMap.exe.
2. Start Microsoft Visual C++.
3. Attach to the ArcMap process using Menu option Build -> Start Debug -> Attach to process.
4. Load the appropriate Visual Basic Source file into the Visual C++ debugger and set the break point.
5. Call the method within ArcMap.

No changes can be made to the source code within the debugger, and variables cannot be inspected, but code execution can be viewed and altered. This is often sufficient to determine what is wrong, especially with logic-related problems.

**UNIT-2**

|  |
| --- |
|  |

|  |  |
| --- | --- |
|  | Variables and Data Types |
|  | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Introduction to Variables | |  |      |  | | --- | | Overview | |  |  |  | | --- | | When you write a program as a time sheet, you may decide that a user will type her weekly hours in one box and her salary in another box; then another box will display her weekly salary. When you are designing the program, you cannot predict the names of the people who will be using the program and you definitely cannot know the weekly hours they will get week after week. What you have to do is ask the computer to create temporary storage areas that one user can use while the program is running. If that box can be used to store a salary, when another user is using the same program, that box should be ready to receive new inputs, new salary for that other user.  The computer memory is made of small storage areas used to hold the things that a program needs while it is running. As a programmer, you specify these things, or you provide them to the computer; the computer then puts them in these storage areas. When you need one of them, you let the computer know. The machine located it and makes it available to you to use as you see fit.  A variable is a value you are ask the computer to store in its memory while the program is running. |      |  | | --- | | Using a Variable | |  |  |  | | --- | | As stated already, 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. |      |  | | --- | | Practical Learning: Using a Variable | |  |  |  |  | | --- | --- | | 1. Start Microsoft Visual Basic and create a new application using Standard EXE 2. Right-click the form and click View Code 3. In the Code Editor, click the arrow of the Object combo box and select Form 4. To use a variable, implement the **Load** event as follows:  |  | | --- | | Private Sub Form\_Load()  SomeColor = vbRed    BackColor = SomeColor  End Sub |  1. Press F5 to test the application. Notice that it appears red 2. After using the form, close it and return to MSVB | |  |

|  |
| --- |
| Variable Declaration |
|  |

|  |  |
| --- | --- |
| Unlike languages referred to as strongly typed, Visual Basic is so flexible you can use any variable just by specifying its name. When you provide this name, the computer directly creates an area in memory for it. Based on this, consider the following code section:   |  | | --- | | Private Sub Form\_Click()  SameColor = vbBlue    SomeColor = vbRed    SumColor = vbRed    BackColor = SameColor  End Sub  Private Sub Form\_KeyDown(KeyCode As Integer, Shift As Integer)  SameColor = vbBlue    SomeColor = vbRed    SumColor = vbGreen    BackColor = SumColor  End Sub  Private Sub Form\_Load()  SameColor = vbBlue    SomeColor = vbRed    SumColor = vbGreen    BackColor = SomeColor  End Sub |   If you execute this program, when the form displays, it would be painted in red. If the user clicks the form, it would be painted in blue. If the user presses a key, the form would be painted in green. There is some confusion in the program. It uses a variable that seems to have a name but initialize three times with different colors. Visual Basic allows you to directly use any name for a variable as you see fit. Fortunately, to eliminate the possibility of this confusion, you can first let Visual Basic know that you will be using a certain variable. Informing Visual Basic about a variable prior to using that variable is referred to as declaring a variable. When a variable has been declared, just like the variable not declared, the computer reserves an area of memory for it.  To declare a variable, type the **Dim** keyword, like this:  Dim  On the right side of Dim, you must type a name for the variable, following the same rules we reviewed above. Here is an example of declaring and using a variable:  Private Sub Form\_Load()  Dim BackgroundColor  BackgroundColor = vbRed    BackColor = BackgroundColor  End Sub  Declaring a variable simply communicates to Visual Basic the name of that variable. You can still use a mix of declared and not-declared variable. This is demonstrated in the following event:  http://www.functionx.com/vb6/windows/project2.gif  Once again, the compiler believes that you are using two variables; one is called BackgroundColor and the other is called SomeColor. This can still create a great deal of confusion because you may be trying to use the same variable referred to twice. The solution to this possible confusion is to tell Visual Basic that a variable cannot be used if it has not been primarily declared. To communicate this, on top of each file you use in the Code Editor, type  Option Explicit  This can also be done automatically for each file by checking the Require Variable Declaration in the Options dialog box. |

|  |
| --- |
| Practical Learning: Using a Variable |
|  |

|  |  |  |
| --- | --- | --- |
| 1. On the main menu of Microsoft Visual Basic, click Tools -> Options... 2. Click the Editor property page. In the Code Settings section, put a check mark in the Require Variable Declaration check box   http://www.functionx.com/vb6/dlgboxes/options2.gif 3. Click OK 4. Close Visual Basic without saving the project 5. Start Microsoft Visual Basic and create a new application using Standard EXE 6. Right-click the form and click View Code 7. In the Code Editor, click the arrow of the Object combo box and select Form 8. To use a variable, implement the **Load** event as follows:  |  | | --- | | http://www.functionx.com/vb6/windows/project3.gif |  1. Press F5 to test the application 2. Notice that you receive an error:   http://www.functionx.com/vb6/windows/error1.gif 3. Click OK 4. On the Standard toolbar, click the End button http://www.functionx.com/vb6/buttons/end1.gif 5. Change the Load event as follows:  |  | | --- | | Private Sub Form\_Load()  Dim BackgroundColor    BackgroundColor = vbRed    BackColor = BackgroundColor  End Sub |  1. After using the form, close it and return to MSVB 2. Click the arrow of the Procedure combo box and select Click    * Implement the events as follows |

|  |
| --- |
| Introduction to Data Types |
|  |

|  |
| --- |
| Introduction |
|  |

|  |
| --- |
| When you decide to use a variable, you are in fact asking the computer to use a certain amount of space to hold that variable. Since different variables will be used for different purposes, you should specify the kind of variable you intend to use, then the computer will figure out how much space is needed for a particular variable. Each variable you use will utilize a certain amount of space in the computer's memory.  Before declaring or using a variable, first decide what kind of role that variable will play in your program. Different variables are meant for different situations. The kind of variable you want to use is referred to as a data type. To specify the kind of variable you want to use, you type the **As** keyword on the right side of the variable's name. The formula to declare such a variable is:  **Dim** *VariableName* **As** *DataType*  Once you know what kind of variable you will need, choose the appropriate data type. Data types are organized in categories such as numbers, characters, or other objects. |

|  |
| --- |
| String |
|  |

|  |
| --- |
| A string is an empty text, a letter, a word or a group of words considered. To declare a string variable, use the String data type. Here is an example: |

Private Sub Form\_Load()

Dim CountryName As String

End Sub

|  |
| --- |
| After declaring the variable, you can initialize. If you want its area of memory to be empty, you can assign it two double-quotes. Here is an example: |

Private Sub Form\_Load()

Dim CountryName As String

CountryName = ""

End Sub

|  |
| --- |
| If you want to store something in the memory space allocated to the variable, assign it a word or group of words included between double-quotes. Here is an example: |

Private Sub Form\_Load()

Dim CountryName As String

CountryName = "Great Britain"

End Sub

|  |
| --- |
| You can also initialize a string variable with another. |

|  |
| --- |
| Boolean |
|  |

|  |
| --- |
| A Boolean variable is one whose value can be only either True or False. To declare such a variable, use the Boolean keyword. Here is an example: |

Private Sub Form\_Load()

Dim IsMarried As Boolean

End Sub

|  |
| --- |
| After declaring a Boolean variable, you can initialize by assigning it either True or False. Here is an example: |

Private Sub Form\_Load()

Dim IsMarried As Boolean

IsMarried = False

End Sub

|  |
| --- |
| Like any other variable, after initializing the variable, it keeps its value until you change its value again. |

|  |
| --- |
| Numeric Data Types |
|  |

|  |
| --- |
| Introduction |
|  |

|  |
| --- |
| A natural number is one that contains only one digit or a combination of digits and no other character, except those added to make it easier to read. Examples of natural numbers are 122, 8, and 2864347. When a natural number is too long, such 3253754343, to make it easier to read, the thousands are separated by a special character. This character depends on the language or group of language and it is called the thousands separator. For US English, this character is the comma. The thousands separator symbol is mainly used only to make the number easier to read.  To support different scenarios, Microsoft provides different types of natural numbers |

|  |
| --- |
| Byte |
|  |

|  |
| --- |
| A byte is a small natural positive number that ranges from 0 to 255. A variable of byte type can be used to hold small values such as a person's age, the number of fingers on an animal, etc.  To declare  a variable for a small number, use the Byte keyword. Here is an example: |

Private Sub Form\_Load()

Dim StudentAge As Byte

End Sub

|  |
| --- |
| Integer |
|  |

|  |
| --- |
| An integer is a natural number larger than the **Byte**. It can hold a value between  -32,768 and 32,767. Examples of such ranges are: the number of pages of a book.  To declare a variable of type integer, use the **Integer** keyword. Here is an example: |

Private Sub Form\_Load()

Dim MusicTracks As Integer

End Sub

|  |
| --- |
| Long Integer |
|  |

|  |
| --- |
| A long integer is a natural number whose value is between –2,147,483,648 and 2,147,483,642. Examples are the population of a city, the distance between places of different countries, the number of words of a book.  To declare a variable that can hold a very large natural number, use the **Long** keyword. Here is an example: |

Private Sub Form\_Load()

Dim Population As Long

End Sub

|  |
| --- |
| Decimal Data Types |
|  |

|  |
| --- |
| Introduction |
|  |

|  |  |
| --- | --- |
| A real number is one that displays a decimal part. This means that the number can be made of two sections separated by a symbol that is referred to as the Decimal Separator or Decimal Symbol. This symbol is different by language, country, group of languages, or group of countries. In US English, this symbol is the period as can be verified from the Regional (and Language) Settings of the Control Panel of computers of most regular users:   |  | | --- | | Regional Options |   On both sides of the Decimal Symbol, digits are used to specify the value of the number. The number of digits on the right side of the symbol determines how much precision the number offers. |

|  |
| --- |
| Single |
|  |

|  |
| --- |
| A single is a decimal number whose value can range from –3.402823e38 and –1.401298e-45 if the number is negative, or 1.401298e-45 and 3.402823e38 if the number is positive.  To declare a variable that can hold small decimal numbers with no concern for precision, use the **Single** data type. Here is an example: |

Private Sub Form\_Load()

Dim CountryName As String

Dim IsMarried As Boolean

Dim StudentAge As Byte

Dim Tracks As Integer

Dim Population As Long

Dim Distance As Single

End Sub

|  |
| --- |
| Double |
|  |

|  |
| --- |
| While the **Single** data type can allow large numbers, it offers less precision. For an even larger number, Microsoft Visual Basic provides the **Double** data type. This is used for a variable that would hold numbers that range from 1.79769313486231e308 to –4.94065645841247e–324 if the number is negative or from 1.79769313486231E308 to 4.94065645841247E–324 if the number is positive.  To declare a variable that can store large decimal numbers with a good level of precision, use the **Double** keyword. |

|  |  |
| --- | --- |
| http://www.functionx.com/buttons/authornote1.gif | In most circumstances, it is preferable to use **Double** instead of **Single** when declaring a variable that would hold a decimal number. Although the **Double** takes more memory spaces (computer memory is not expensive anymore(!)), it provides more precision. |

Here is an example of declaring a **Double** variable:

Private Sub Form\_Load()

Dim Distance As Double

End Sub

|  |
| --- |
| Currency |
|  |

|  |
| --- |
| The **Currency** data type is used for a variable that can hold monetary values. To declare such a variable, use the **Currency** keyword. Here is an example: |

Private Sub Form\_Load()

Dim CountryName As String

Dim IsMarried As Boolean

Dim StudentAge As Byte

Dim Tracks As Integer

Dim Population As Long

Dim Distance As Single

Dim StartingSalary As Currency

End Sub

|  |
| --- |
| Other Data Types |
|  |

|  |
| --- |
| Date |
|  |

|  |
| --- |
| A date is a numeric value that represents the number of days that have elapsed since a determined period. A time is a numeric value that represents the number of seconds that have elapsed in a day.  To declare a variable that can hold either date values, time values, or both, use the **Date** keyword. After the variable has been declared, you will configure it to the appropriate value. Here are two examples: |

Private Sub Form\_Load()

Dim CountryName As String

Dim IsMarried As Boolean

Dim StudentAge As Byte

Dim Tracks As Integer

Dim Population As Long

Dim Distance As Single

Dim StartingSalary As Currency

Dim DateOfBirth As Date

Dim KickOffTime As Date

End Sub

|  |
| --- |
| Object |
|  |

|  |
| --- |
| An **Object** is almost anything else that you want to use in your program. If you don't specify a data type or can't figure out what data type you want to use, you can use the **Variant** keyword or let Visual Basic use the **Variant** data type. |

|  |
| --- |
| Variant |
|  |

|  |
| --- |
| A **Variant** can be used to declare any kind of variable. You can use a variant when you can't make up your mind regarding a variable but, as a beginning programmer, you should avoid it.  Here is a table of various data types and the [amount of memory space](http://www.functionx.com/references/numsystem.htm) each one uses: |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Data type** | **Description** | **Range** | | **Byte** | 1-byte binary data | 0 to 255 | | **Integer** | 2-byte integer | – 32,768 to 32,767 | | **Long** | 4-byte integer | – 2,147,483,648 to 2,147,483,647 | | **Single** | 4-byte floating-point number | – 3.402823E38 to – 1.401298E – 45 (negative values) | | 1.401298E – 45 to 3.402823E38 (positive values) | | **Double** | 8-byte floating-point number | – 1.79769313486231E308 to – 4.94065645841247E – 324 (negative values) | | 4.94065645841247E – 324 to 1.79769313486231E308 (positive values) | | **Currency** | 8-byte number with fixed decimal point | – 922,337,203,685,477.5808 to 922,337,203,685,477.5807 | | **String** | String of characters | Zero to approximately two billion characters | | **Variant** | Date/time, floating-point number, integer, string, or object.  16 bytes, plus 1 byte for each character if a string value. | Date values: January 1, 100 to December 31, 9999  Numeric values: same range as **Double**  String values: same range as **String** Can also contain **Error** or **Null** values | | **Boolean** | 2 bytes | **True** or **False** | | **Date** | 8-byte date/time value | January 1, 100 to December 31, 9999 | | **Object** | 4 bytes | Any **Object** reference | |

|  |
| --- |
| Using Variables |
|  |

|  |
| --- |
| Details on Declaring Variables |
|  |

|  |
| --- |
| We have learned how to declare a variable as follows:  Dim CountryName As String  We also saw that we can declare different variables each on its own line as follows;  **Dim** FirstName **As** String  **Dim** LastName **As** String  **Dim** Salary **As** Currency  **Dim** AlreadyVisited **As** Boolean  If you have many variables of the same data type, you can declare them on the same line, each separated with a comma. Remember to specify their common type. Here is an example:  Private Sub Form\_Load()  **Dim CountryName, Address, City, State As String**  Dim IsMarried As Boolean  Dim StudentAge As Byte  Dim Tracks As Integer  Dim Population As Long  Dim Distance As Single  **Dim StartingSalary, WeeklyEarnings As Currency**  **Dim DateOfBirth, KickOffTime As Date**  End Sub  When naming your variables, besides the above suggestions, you can start a variable's name with one to three-letter prefix that could identify the data type used. Here are a few suggestions: |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Data Type** | **Prefix** | **Example** | | Boolean | bln | blnFound | | Byte | byt | bytTracks | | Date/Time | dtm | dteStartOfShift | | Double | dbl | dblDistance | | Error | err | errCantOpen | | Integer | int | intNbrOfStudents | | Long | lng | lngPopulation | | Object | obj | objConnection | | Single | sng | sngAge | | String | str | strCountryName | | Currency | cur | curHourlySalary | | Variant | var | varFullName | |

|  |
| --- |
| Constants |
|  |

|  |
| --- |
| Introduction |
|  |

|  |
| --- |
| A constant is a value that doesn't change. There are two types of constants you will use in your programs: those supplied to you and those you define yourself. |

|  |
| --- |
| The Carriage Return-Line Feed Constant |
|  |

|  |
| --- |
| Visual Basic provides the **vbCrLf** constant. It is used to interrupt a line of code and move to the next line. |

|  |
| --- |
| Built-in Constants: PI |
|  |

|  |
| --- |
| PI is a mathematical constant whose value is approximately equal to 3.1415926535897932. It is highly used in operations that involve circles or geometric variants of a circle: cylinder, sphere, cone, etc. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Forcing Declaration of Variables (Option Explicit)Why you should Declare All Variables The example on the previous page would run just as well if you didn't declare your variables:  Example with varaibles commented out  *If you comment out the variable declarations like this, your macro will still work.*    In the above case, VBA will reach the first two lines creating variables:  HeroName = Range("C4")  HeroRating = Range("C6")  Because you haven't explicitly declared **HeroName** and **HeroRating**, VBA will create them on the fly for you, giving each the type **Variant**.    So far, so good?  Well, yes - until you mistype a variable name.  Here's what would happen for a simple mistyping:   |  |  | | --- | --- | | *Misprinted VBA code* | *Superhero wrongly entered* | | *There's a misprint here - how quickly can you see it?* | *The superhero you get as a result* |   *The problem above could be avoided so easily, if you could just tell VBA that wherever you use a variable, you have to declare it.  Read on!* Forcing Declaration of Variables The solution to the above problem is to include two magic words at the top of each module - **Option Explicit**:  Option Explicit  Sub RecordVote()  'the name of each superhero and their rating  'Dim HeroName As String  'the rating assigned to them  'Dim HeroRating As Long  'go to the votes sheet and get the value of the superhero, and their rating  Worksheets("Votes").Select  HeroNane = Range("C4").Value  HeroRating = Range("C6").Value  Here's what you get if you run the above macro now:  Example of variable not defined message  *The result of running the macro.  VBA highlights the variable you've misspelt, and tells you that it's not defined.*   Setting the Option to Require Variables as a Default Given that forcing yourself to declare all variables like this is such a good idea, how can you avoid having to type in **Option Explicit** at the top of every new module you create?  Here's how!   1. From the menu within VBA (not Excel) select **Tools** -> **Options**. 2. Tick the **Require Variable Declaration** box shown below. 3. Select OK.   Require variables tick box  *Tick the box shown!*  From now on, every time you create a new module it will have the words **Option Explicit** at the top, and you'll have to declare any variables that you use. Operators and Expressions in Visual Basic An operator is a code element that performs an operation on one or more code elements that hold values. Value elements include variables, constants, literals, properties, returns from Function and Operator procedures, and expressions.  An expression is a series of value elements combined with operators, which yields a new value. The operators act on the value elements by performing calculations, comparisons, or other operations. [Types of Operators](javascript:void(0))  Visual Basic provides the following types of operators:   * [Arithmetic Operators](https://msdn.microsoft.com/en-us/library/b6ex274z.aspx) perform familiar calculations on numeric values, including shifting their bit patterns. * [Comparison Operators](https://msdn.microsoft.com/en-us/library/215yacb6.aspx) compare two expressions and return a Boolean value representing the result of the comparison. * [Concatenation Operators](https://msdn.microsoft.com/en-us/library/te2585xw.aspx) join multiple strings into a single string. * [Logical and Bitwise Operators in Visual Basic](https://msdn.microsoft.com/en-us/library/wz3k228a.aspx) combine Boolean or numeric values and return a result of the same data type as the values.   The value elements that are combined with an operator are called operands of that operator. Operators combined with value elements form expressions, except for the assignment operator, which forms a statement. For more information, see [Statements in Visual Basic](https://msdn.microsoft.com/en-us/library/865x40k4.aspx). [Evaluation of Expressions](javascript:void(0))  The end result of an expression represents a value, which is typically of a familiar data type such as Boolean, String, or a numeric type.  The following are examples of expressions.  5 + 4  ' The preceding expression evaluates to 9.  15 \* System.Math.Sqrt(9) + x  ' The preceding expression evaluates to 45 plus the value of x.  "Concat" & "ena" & "tion"  ' The preceding expression evaluates to "Concatenation".  763 < 23  ' The preceding expression evaluates to False.  Several operators can perform actions in a single expression or statement, as the following example illustrates.  VB  [Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_7953e305-d294-4623-8324-7e310618d0a0');" \o "Copy to clipboard.)  x = 45 + y \* z ^ 2  In the preceding example, Visual Basic performs the operations in the expression on the right side of the assignment operator (=), then assigns the resulting value to the variable x on the left. There is no practical limit to the number of operators that can be combined into an expression, but an understanding of [Operator Precedence in Visual Basic](https://msdn.microsoft.com/en-us/library/fw84t893.aspx) is necessary to ensure that you get the results you expect.  An operator is a special symbol which indicates a certain process is carried out. Operators in programming languages are taken from mathematics. Programmers work with data. The operators are used to process data.  We have several types of operators:   * Arithmetic operators * Boolean operators * Relational operators * Bitwise operators   An operator may have one or two operands. An operand is one of the inputs (arguments) of an operator. Those operators that work with only one operand are called unary operators. Those who work with two operands are called binary operators.  Option Strict On  Module Example  Sub Main()  Console.WriteLine(2)  Console.WriteLine(-2)  Console.WriteLine(2+2)  Console.WriteLine(2-2)  End Sub  End Module  The + and - signs can be addition and subtraction operators as well as unary sign operators. It depends on the situation.  Option Strict On  Module Example  Dim a As Byte  Sub Main()  a = 1  Console.WriteLine(-a) ' Prints -1  Console.WriteLine(-(-a)) ' Prints 1  End Sub  End Module  The plus sign can be used to indicate that we have a positive number. But it is mostly not used. The minus sign changes the sign of a value.  Option Strict On  Module Example  Dim a As Byte  Dim b As Byte  Sub Main()  a = 3 \* 3  b = 2 + 2    Console.WriteLine(a) ' Prints 9  Console.WriteLine(b) ' Print 4  End Sub  End Module  Multiplication and addition operators are examples of binary operators. They are used with two operands. The assignment operator The assignment operator = assigns a value to a variable. A variable is a placeholder for a value. In mathematics, the = operator has a different meaning. In an equation, the = operator is an equality operator. The left side of the equation is equal to the right one.  x = 1  Console.WriteLine(x) ' Prints 1  Here we assign a number to the x variable.  x = x + 1  Console.WriteLine(x)  The previous expression does not make sense in mathematics. But it is legal in programming. The expression adds 1 to the x variable. The right side is equal to 2 and 2 is assigned to x.  3 = x  This code example results in syntax error. We cannot assign a value to a literal. Arithmetic operators The following is a table of arithmetic operators in Visual Basic.   |  |  | | --- | --- | | **Symbol** | **Name** | | + | Addition | | - | Subtraction | | \* | Multiplication | | / | Division | | \ | Integer Division | | Mod | Modulo | | ^ | Exponentiation |   The following example shows arithmetic operations.  Option Strict On  Module Example  Dim a As Byte  Dim b As Byte  Dim c As Byte  Dim add As Byte  Dim sb As Byte  Dim mult As Byte  Dim div As Byte  Sub Main()  a = 10  b = 11  c = 12  add = a + b + c  sb = c - a  mult = a \* b  div = CType(c / 3, Byte)    Console.WriteLine(add)  Console.WriteLine(sb)  Console.WriteLine(mult)  Console.WriteLine(div)  End Sub  End Module  In the preceding example, we use addition, subtraction, multiplication and division operations. This is all familiar from the mathematics.  $ ./arithmetic.exe  33  2  110  4  Output of the example.  Next we will show the distinction between normal and integer division.  Option Strict On  Module Example  Dim a As Single = 5  Dim b As Single = 2  Dim c As Single  Sub Main()  c = 5 / 2  Console.WriteLine(c)  c = 5 \ 2  Console.WriteLine(c)    End Sub  End Module  In the preceding example, we divide two numbers using normal and integer division operator. Visual Basic has two distinct operators for division.  Dim a As Single = 5  We use floating point data types.  c = 5 / 2  Console.WriteLine(c)  This is the 'normal' division operation. It returns 2.5, as expected.  c = 5 \ 2  Console.WriteLine(c)  This is integer division. The result of this operation is always and integer. The c variable has value 2.  $ ./division.exe  2.5  2  Result of the division.exe program.  The last two operators that we will mention are modulo operator and exponentiation operator.  Console.WriteLine(9 Mod 4) ' Prints 1  The Mod operator is called the modulo operator. It finds the remainder of division of one number by another. 9 Mod 4, 9 modulo 4 is 1, because 4 goes into 9 twice with a remainder of 1. Modulo operator can be handy for example when we want to check for prime numbers.  Finally, we will mention exponentiation operator.  Console.WriteLine(9 ^ 2) ' Prints 81  9 ^ 2 = 9 \* 9 = 81 Concatenating strings In Visual Basic we have two operators for string concatenation. The plus + operator and the & ampersand operator.  Option Strict On  Module Example  Sub Main()  Console.WriteLine("Return " & "of " & "the king")  Console.WriteLine("Return " + "of " + "the king")  End Sub  End Module  We join three strings together using both operators.  $ ./concatstrings.exe  Return of the king  Return of the king  And this is, what we get. Same result for both cases. Boolean operators In Visual Basic, we have the following logical operators. Boolean operators are also called logical.   |  |  | | --- | --- | | **Symbol** | **Name** | | And | logical conjunction | | AndAlso | short circuit And | | Or | logical inclusion | | OrElse | short circuit Or | | Xor | logical inclusion | | Not | negation |   Boolean operators are used to work with truth values.  Option Strict On  Module Example  Dim x As Byte = 3  Dim y As Byte = 8  Sub Main()  Console.WriteLine(x = y)  Console.WriteLine(y > x)  If (y > x)  Console.WriteLine("y is greater than x")  End If  End Sub  End Module  Many expressions result in a boolean value. Boolean values are used in conditional statements.  Console.WriteLine(x = y)  Console.WriteLine(y > x)  Relational operators always result in a Boolean value. These two lines print False and True.  If (y > x)  Console.WriteLine("y is greater than x")  End If  The body of the If statement is executed only if the condition inside the parentheses is met. The x > y returns True, so the message "y is greater than x" is printed to the terminal.  Option Strict On  Module Example  Dim a As Boolean  Dim b As Boolean  Dim c As Boolean  Dim d As Boolean  Sub Main()  a = (True And True)  b = (True And False)  c = (False And True)  d = (False And False)  Console.WriteLine(a)  Console.WriteLine(b)  Console.WriteLine(c)  Console.WriteLine(d)  End Sub  End Module  Example shows the logical And operator. It evaluates to True only if both operands are True.  $ ./andop.exe  True  False  False  False  The logical Xor operator evaluates to True, if exactly one of the operands is True.  Option Strict On  Module Example  Dim a As Boolean  Dim b As Boolean  Dim c As Boolean  Dim d As Boolean  Sub Main  a = (True Xor True)  b = (True Xor False)  c = (False Xor True)  d = (False Xor False)  Console.WriteLine(a)  Console.WriteLine(b)  Console.WriteLine(c)  Console.WriteLine(d)  End Sub  End Module  The logical Xor evaluates to False if both operands are True or both False.  $ ./xorop.exe  False  True  True  False  The logical Or operator evaluates to True, if either of the operands is True.  Option Strict On  Module Example  Sub Main()  Dim a As Boolean = True Or True  Dim b As Boolean = True Or False  Dim c As Boolean = False Or True  Dim d As Boolean = False Or False  Console.WriteLine(a)  Console.WriteLine(b)  Console.WriteLine(c)  Console.WriteLine(d)  End Sub  End Module  If one of the sides of the operator is True, the outcome of the operation is True.  $ ./orop.exe  True  True  True  False  The negation operator Not makes True False and False True.  Option Strict On  Module Example  Sub Main()  Console.WriteLine(Not True)  Console.WriteLine(Not False)  Console.WriteLine(Not (4 < 3))  End Sub  End Module  The example shows the negation operator in action.  $ ./negation.exe  False  True  True  AndAlso, OrElse operators are short circuit evaluated. Short circuit evaluation means that the second argument is only evaluated if the first argument does not suffice to determine the value of the expression: when the first argument of And evaluates to false, the overall value must be false; and when the first argument of Or evaluates to true, the overall value must be true. (wikipedia) Short circuit evaluation is used mainly to improve performance.  An example may clarify this a bit more.  Option Strict On  Module Example  Sub Main()  Console.WriteLine("Short circuit")  If (one AndAlso two)  Console.WriteLine("Pass")  End If  Console.WriteLine("#############")  If (one And two)  Console.WriteLine("Pass")  End If  End Sub  Function one As Boolean  Console.WriteLine("Inside one")  Return False  End Function  Function two As Boolean  Console.WriteLine("Inside two")  Return True  End Function  End Module  We have two functions in the example. Functions, unlike subroutines, return values. This is the main difference between them.  If (one AndAlso two)  Console.WriteLine("Pass")  End If  The one function returns False. The short circuit AndAlso does not evaluate the second function. It is not necessary. Once an operand is False, the result of the logical conclusion is always False. Only "Inside one" is printed to the console.  Console.WriteLine("#############")  If (one And two)  Console.WriteLine("Pass")  End If  In the second case, we use the And. In this case, both functions are called. Even though it is not necessary for the result of the expression.  $ ./shorcircuit.exe  Short circuit  Inside one  #############  Inside one  Inside two  Result of the shorcircuit.exe program. Relational Operators Relational operators are used to compare values. These operators always result in a boolean value.   |  |  | | --- | --- | | **Symbol** | **Meaning** | | < | less than | | <= | less than or equal to | | > | greater than | | >= | greater than or equal to | | == | equal to | | <> | not equal to | | Is | compares references |   Relational operators are also called comparison operators.  Console.WriteLine(3 < 4) ' Prints True  Console.WriteLine(3 = 4) ' Prints False  Console.WriteLine(4 >= 3) ' Prints True  As we already mentioned, the relational operators return boolean values. Note that in Visual Basic, the comparison operator is =. Not == like in C and C influenced languages.  Notice that the relational operators are not limited to numbers. We can use them for other objects as well. Although they might not always be meaningful.  Option Strict On  Module Example  Sub Main()  Console.WriteLine("six" = "six") ' Prints True  ' Console.WriteLine("a" > 6) this would throw  ' an exception  Console.WriteLine("a" < "b") ' Prints True  End Sub  End Module  We can compare string objects too. Comparison operators in a string context compare the sorting order of the characters.  Console.WriteLine("a" < "b") ' Prints True  What exactly happens here? Computers do not know characters or strings. For them, everything is just a number. Characters are special numbers stored in specific tables. Like ASCII.  Option Strict On  Module Example  Sub Main()  Console.WriteLine("a" < "b")    Console.WriteLine("a is: {0}", Asc("a"))  Console.WriteLine("b is: {0}", Asc("b"))    End Sub  End Module  Internally, the a and b characters are numbers. So when we compare two characters, we compare their stored numbers. The built-in Asc function returns the ASCII value of a single character.  $ ./compare.exe  True  a is: 97  b is: 98  In fact, we compare two numbers, 97 with 98.  Console.WriteLine("ab" > "aa") ' Prints True  Say we have a string with more characters. If the first characters are equal, we compare the next ones. In our case, the b character at the second position has a greater value than the a character. That is why "ab" string is greater than "aa" string. Comparing strings in such a way does not make much sense, of course. But it is technically possible.  Finally, we will mention the Is operator. The operator checks if two object references refer to the same object. It does not perform value comparisons.  Option Strict On  Module Example  Sub Main()  Dim o1 As Object = New Object  Dim o2 As Object = New Object  Dim o3 As Object    o3 = o2  Console.WriteLine(o1 Is o2)  Console.WriteLine(o3 Is o2)  End Sub  End Module  We create three objects and compare them with the Is operator.  Dim o1 As Object = New Object  Dim o2 As Object = New Object  We declare and initialise two Object instances. The Object class is a base class for all classes in the .NET framework. We will describe it later in more detail.  Dim o3 As Object  The third variable is only declared.  o3 = o2  The o3 now refers to the o2 object. They are two references to the same object.  Console.WriteLine(o1 Is o2)  Console.WriteLine(o3 Is o2)  In the first case, we get False. The o1 and o2 are two different objects. In the second case, we get True. The o3 and o2 refer to the same object. Bitwise operators Decimal numbers are natural to humans. Binary numbers are native to computers. Binary, octal, decimal, or hexadecimal symbols are only notations of the same number. Bitwise operators work with bits of a binary number. Bitwise operators are seldom used in higher level languages like Visual Basic.   |  |  | | --- | --- | | **Symbol** | **Meaning** | | Not | bitwise negation | | Xor | bitwise exclusive or | | And | bitwise and | | Or | bitwise or |   The bitwise negation operator changes each 1 to 0 and 0 to 1.  Console.WriteLine(Not 7) ' Prints -8  Console.WriteLine(Not -8) ' Prints 7  The operator reverts all bits of a number 7. One of the bits also determines, whether the number is negative or not. If we negate all the bits one more time, we get number 7 again.  The bitwise and operator performs bit-by-bit comparison between two numbers. The result for a bit position is 1 only if both corresponding bits in the operands are 1.  00110  And 00011  = 00010  The first number is a binary notation of 6, the second is 3 and the result is 2.  Console.WriteLine(6 And 3) ' Prints 2  Console.WriteLine(3 And 6) ' Prints 2  The bitwise or operator performs bit-by-bit comparison between two numbers. The result for a bit position is 1 if either of the corresponding bits in the operands is 1.  00110  Or 00011  = 00111  The result is 00110 or decimal 7.  Console.WriteLine(6 Or 3) ' Prints 7  Console.WriteLine(3 Or 6) ' Prints 7  The bitwise exclusive or operator performs bit-by-bit comparison between two numbers. The result for a bit position is 1 if one or the other (but not both) of the corresponding bits in the operands is 1.  00110  Xor 00011  = 00101  The result is 00101 or decimal 5.  Console.WriteLine(6 Xor 3) ' Prints 5 Compound assignment operators The compound assignment operators consist of two operators. They are shorthand operators.  Option Strict On  Module Example  Dim a As Integer  Sub Main  a = 1  a = a + 1  Console.WriteLine(a) ' Prints 2  a += 1  Console.WriteLine(a) ' Prints 3  End Sub  End Module  The += compound operator is one of these shorthand operators. They are less readable than the full expressions but experienced programmers often use them.  Other compound operators are:  -= \*= \= /= &= ^= Operator precedence The operator precedence tells us which operators are evaluated first. The precedence level is necessary to avoid ambiguity in expressions.  What is the outcome of the following expression, 28 or 40?  3 + 5 \* 5  Like in mathematics, the multiplication operator has a higher precedence than addition operator. So the outcome is 28.  (3 + 5) \* 5  To change the order of evaluation, we can use parentheses. Expressions inside parentheses are always evaluated first.  The following list shows common Visual Basic operators ordered by precedence (highest precedence first):   |  |  | | --- | --- | | **Operator(s)** | **Description** | | ^ | exponentiation | | + - | unary identity and negation | | \* / | multiplication, float division | | \ | integer division | | Mod | modulus | | + - | addition, subtraction, string concatenation | | & | string concatenation | | << >> | arithmetic bit shift | | = <> < > >= <= Is IsNot Like TypeOf Is | All comparison operators | | Not | negation | | And AndAlso | conjunction | | Or OrElse | Inclusive disjunction | | Xor | Exclusive disjunction |   Operators on the same line in the list have the same precedence.  Option Strict On  Module Example  Sub Main()  Console.WriteLine(3 + 5 \* 5)  Console.WriteLine((3 + 5) \* 5)  Console.WriteLine(Not True Or True)  Console.WriteLine(Not (True Or True))  End Sub  End Module  In this code example, we show some common expressions. The outcome of each expression is dependent on the precedence level.  Console.WriteLine(3 + 5 \* 5)  This line prints 28. The multiplication operator has a higher precedence than addition. First the product of 5\*5 is calculated. Then 3 is added.  Console.WriteLine(Not True Or True)  In this case, the negation operator has a higher precedence. First, the first True value is negated to False, than the Or operator combines False and True, which gives True in the end.  $ ./precedence.exe  28  40  True  False Associativity Sometimes the precedence is not satisfactory to determine the outcome of an expression. There is another rule called associativity. The associativity of operators determines the order of evaluation of operators with the same precedence level.  9 / 3 \* 3  What is the outcome of this expression? 9 or 1? The multiplication, deletion and the modulo operator are left to right associated. So the expression is evaluated this way: (9 / 3) \* 3 and the result is 9.  Arithmetic, boolean, relational and bitwise operators are all left o right associated.  On the other hand, the assignment operator is right associated.  a = b = c = d = 0  Console.WriteLine("{0} {1} {2} {3}", a, b, c, d) ' Prints 0 0 0 0  If the association was left to right, the previous expression would not be possible.  The compound assignment operators are right to left associated.  j = 0  j \*= 3 + 1  Console.WriteLine(j)  You might expect the result to be 1. But the actual result is 0. Because of the associativity. The expression on the right is evaluated first and than the compound assignment operator is applied. AddressOf operator The AddressOf operator creates a function delegate that points to another function. Delegates are type safe function pointers, they are used to call methods of other objects.  Option Strict On  Module Example  Delegate Sub Display  Dim msg As New Display(AddressOf Message1)  Sub Main()  msg.Invoke()  msg = New Display(AddressOf Message2)  msg.Invoke()  End Sub  Sub Message1()  Console.WriteLine("This is message 1")  End Sub  Sub Message2()  Console.WriteLine("This is message 2")  End Sub  End Module  In the code example, we use the AddressOf operator to point to two different subroutines.  Delegate Sub Display  We need to declare a delegate.  Dim msg As New Display(AddressOf Message1)  The delegate takes the address of a subroutine using the AddressOf operator. Now we have a type-safe pointer to the Message1() subroutine.  msg.Invoke()  The Invoke() method calls the method, to which the delegate points.  msg = New Display(AddressOf Message2)  msg.Invoke()  Now we give the delegate an address of another subroutine.  $ ./addressof.exe  This is message 1  This is message 2  Both messages are printed to the console.  **Input\Output in VB Using A Data File**  **Introduction**  You may want to read some data from an input file and write results into  another output file. In these cases, it is useful to use a plain “text file” (“\*.txt”)  which includes no information other than plain text. Files for other software  (e.g., MS Word) contain complicated data that are used to format the  information in the file.  Below is a file containing the names of a few dead rock stars and the dates they  died. You may not have realized that Brian Jones (original rhythm guitarist for  the Rolling Stones) and Jim Morrison died two years apart on the same day.  **Access Visual Basic**  1. Open VB from the “start” menu, that is:  **Start\All Programs\Microsoft Visual Basic**  2. Select “**File**  **New Project**” to create a new project  3. Select “**Windows Form Application**”. You can give your “application” a  name, or simply accept the default offered “**WindowsApplication1**”.  Click “**OK**”.  **Prepare the Graphic User Interface**  Develop the "form" shown in the figure above that we will use in the project.  The form should contain a label, a text box, two command buttons, and a  picture box with the following properties, and look like:  Lab 4 - CE 311 K - McKinney  2  **Add VB Code to the Project**  Let’s add some code to the “Stop” button. Double click on the “Stop” button  and type in the following code:  Private Sub Button2\_Click(…) Handles Button2.Click  End  End Sub  Double click on the “Go!” button and type in the following code:  Private Sub Button1\_Click(…) Handles Button1.Click  Dim sr As IO.StreamReader = IO.File.OpenText("C:\temp\name.txt")  Dim Line, Text(), lastName, firstName, birth As String  Line = sr.ReadLine  Text = Line.Split(" ")  lastName = Text(0)  firstName = Text(1)  birth = Text(2)  ListBox1.Items.Add(firstName & " " & lastName & " " & ", Born: " & birth)  Line = sr.ReadLine  Text = Line.Split(" ")  lastName = Text(0)  firstName = Text(1)  birth = Text(2)  ListBox1.Items.Add(firstName & " " & lastName & " " & ", Born: " & birth)  End Sub  Lab 4 - CE 311 K - McKinney  3  Now, test your code. You should get something that looks like the following.  Let's talk about some of the code here:  Dim sr As IO.StreamReader = IO.File.OpenText("C:\temp\name.txt")  Variable “sr”:  “sr” is declared to be an “IO” object that is capable of receiving input to  your program (it can “read streams” of input).  “sr” is assigned to the “IO” object that points to the file  "C:\temp\name.txt".  Input File Directory Path and Name:  "C:\temp\CE311K\InputOutput\ " is the directory path and “name.txt“ is  the name of the file to be opened. In this example, the path is  “hardcoded” into the program. Be sure that the path is correct for the  location of your file "name.txt".  Dim:  Dim Line, Text(), lastName, firstName, birth As String  The variables “Line”, “Text”, “lastName”, “firstName”, and “birth” are  declared to be “string” variables. “Text” is somewhat special as you can  tell by the parentheses “()” after the name. This means that it is an  “array” variable and can be broken into pieces; we’ll cut it into three  pieces, “Text(0)”, “Text(1)”, and “Text(2)”.  Lab 4 - CE 311 K - McKinney  4  ReadLine:  Line = sr.ReadLine  This line reads one line from the file assigned to the variable “sr” and  assigns that string to the variable “Line”.  Split:  Text = Line.Split(" ")  This line “splits” the variable “Line” into pieces. The splits are made at  locations in the string where the character in the double quotes appear, a  blank space in this case “ “. The pieces are assigned to the variable  “Text”.  Now that we have split the line we read into pieces, we can assign the pieces to  variables:  lastName = Text(0)  firstName = Text(1)  birth = Text(2)  Here we put the first piece “Text(0)” into the variable “lastName”, and so  forth.  Output:  ListBox1.Items.Add(firstName & " " & lastName & " " & ", Born: " & birth)  This adds the indicated output to the listbox “ListBox1”  Lab 4 - CE 311 K - McKinney  5  **Write Data to an Output File**  Next, let’s write the output to another file.  Private Sub Button1\_Click(…) Handles Button1.Click  Dim sr As IO.StreamReader = IO.File.OpenText("C:\temp\name.txt")  Dim sw As IO.StreamWriter = IO.File.CreateText("C:\temp\output.txt")  Dim Line, Text(), lastName, firstName, birth As String  sw.WriteLine("This is the output file.")  Line = sr.ReadLine  Text = Line.Split(" ")  lastName = Text(0)  firstName = Text(1)  birth = Text(2)  ListBox1.Items.Add(firstName & " " & lastName & " " & ", Born: " & birth)  sw.WriteLine(firstName & " " & lastName & " " & ", Born: " & birth)  Line = sr.ReadLine  Text = Line.Split(" "c)  lastName = Text(0)  firstName = Text(1)  birth = Text(2)  ListBox1.Items.Add(firstName & " " & lastName & " " & ", Born: " & birth)  sw.WriteLine(firstName & " " & lastName & " " & ", Born: " & birth)  sw.Close()  End Sub  Let's talk about some of the code here:  Dim sw As IO.StreamWriter = IO.File.CreateText("C:\temp\output.txt")  Variable “sw”:  “sw” is declared to be an “IO” object that is capable of receiving output  from your program (it can “write streams” of output).  “sw” is assigned to the “IO” object that points to the file  "C:\temp\output.txt".  Lab 4 - CE 311 K - McKinney  6  WriteLine:  sw.WriteLine(“This is the output file”)  This line writes one line into the file assigned to the variable “sw”.  sw.WriteLine(firstName & " " & lastName & " " & ", Born: " & birth)  This line writes one line into the output file with the variables indicated.  Close:  sw.Close()  This line closes the output file and allows us to look at the file contents  after the program is stopped.  Lab 4 - CE 311 K - McKinney  7  **Assignment**  Write a VB code with graphic user interface to find the average annual  temperature in Austin. Use the program “Notebook” to create a text file to use  as input for your program. The input file should contain the information in the  following table: one column for the name of the month and one column for the  temperature of that month. Your program should open the input file and read  it one line at a time and compute the average of the monthly temperatures to  get the annual average temperature. Write the result to an output file.  Month Temperature (deg F)  January 58  February 63  March 71  April 79  May 84  June 91  July 95  August 95  September 90  October 82  November 71  December 62  **To be turned in:**  **You can paste all of the following items into a MS Word document and**  **print it and hand it in, or email it to me at:** daene@aol.com**. Be sure to put**  **you name on the first line of the document.**  1. A printout of he input file from your program.  2. A printout of the VB code used in your program.  3. A printout of the output file from your program.  **Unit-3**  **Programming with vb :-Decisions and conditions in vb** |

# Decision Structures (Visual Basic)

**Visual Studio 2015**

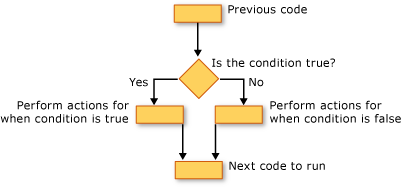
[Other Versions](javascript:;)

https://i-msdn.sec.s-msft.com/Areas/Epx/Content/Images/ImageSprite.png?v=635777873432767956

* [Visual Studio 2013](https://msdn.microsoft.com/en-us/library/hh892482(v=vs.120).aspx)
* [Visual Studio 2012](https://msdn.microsoft.com/en-us/library/hh892482(v=vs.110).aspx)
* [Visual Studio 2010](https://msdn.microsoft.com/en-us/library/hh892482(v=vs.100).aspx)
* [Visual Studio 2008](https://msdn.microsoft.com/en-us/library/hh892482(v=vs.90).aspx)
* [Visual Studio 2005](https://msdn.microsoft.com/en-us/library/hh892482(v=vs.80).aspx)
* [Visual Studio .NET 2003](https://msdn.microsoft.com/en-us/library/hh892482(v=vs.71).aspx)

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.

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


## [If...Then...Else Construction](javascript:void(0))

If...Then...Else constructions let you test for one or more conditions and run one or more statements depending on each condition. You can test conditions and take actions in the following ways:

* Run one or more statements if a condition is True
* Run one or more statements if a condition is False
* Run some statements if a condition is True and others if it is False
* Test an additional condition if a prior condition is False

The control structure that offers all these possibilities is the [If...Then...Else Statement (Visual Basic)](https://msdn.microsoft.com/en-us/library/752y8abs.aspx). You can use a single-line version if you have just one test and one statement to run. If you have a more complex set of conditions and actions, you can use the multiple-line version.

## [Select...Case Construction](javascript:void(0))

The Select...Case construction lets you evaluate an expression one time and run different sets of statements based on different possible values. For more information, see [Select...Case Statement (Visual Basic)](https://msdn.microsoft.com/en-us/library/cy37t14y.aspx).

## [Try...Catch...Finally Construction](javascript:void(0))

Try...Catch...Finally constructions let you run a set of statements under an environment that retains control if any one of your statements causes an exception. You can take different actions for different exceptions. You can optionally specify a block of code that runs before you exit the whole Try...Catch...Finally construction, regardless of what occurs. For more information, see [Try...Catch...Finally Statement (Visual Basic)](https://msdn.microsoft.com/en-us/library/fk6t46tz.aspx).

|  |
| --- |
| **NoteNote** |
| For many control structures, when you click a keyword, all of the keywords in the structure are highlighted. For instance, when you click If in an If...Then...Else construction, all instances of If, Then, ElseIf, Else, and End If in the construction are highlighted. To move to the next or previous highlighted keyword, press CTRL+SHIFT+DOWN ARROW or CTRL+SHIFT+UP ARROW. |

# Loop Structures (Visual Basic)

**Visual Studio 2015**

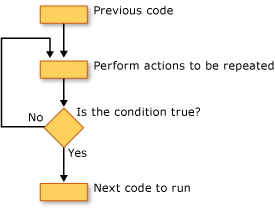
[Other Versions](javascript:;)

https://i-msdn.sec.s-msft.com/Areas/Epx/Content/Images/ImageSprite.png?v=635777873432767956

* [Visual Studio 2013](https://msdn.microsoft.com/en-us/library/ezk76t25(v=vs.120).aspx)
* [Visual Studio 2012](https://msdn.microsoft.com/en-us/library/ezk76t25(v=vs.110).aspx)
* [Visual Studio 2010](https://msdn.microsoft.com/en-us/library/ezk76t25(v=vs.100).aspx)
* [Visual Studio 2008](https://msdn.microsoft.com/en-us/library/ezk76t25(v=vs.90).aspx)
* [Visual Studio 2005](https://msdn.microsoft.com/en-us/library/ezk76t25(v=vs.80).aspx)
* [Visual Studio .NET 2003](https://msdn.microsoft.com/en-us/library/ezk76t25(v=vs.71).aspx)

Visual Basic loop structures allow you to run one or more lines of code repetitively. You can repeat the statements in a loop structure until a condition is **True**, until a condition is **False**, a specified number of times, or once for each element in a collection.

The following illustration shows a loop structure that runs a set of statements until a condition becomes true.



Running a set of statements until a condition becomes true

## [While Loops](javascript:void(0))

The **While**...**End While** construction runs a set of statements as long as the condition specified in the **While** statement is **True**. For more information, see [While...End While Statement (Visual Basic)](https://msdn.microsoft.com/en-us/library/zh1f56zs.aspx).

## [Do Loops](javascript:void(0))

The **Do**...**Loop** construction allows you to test a condition at either the beginning or the end of a loop structure. You can also specify whether to repeat the loop while the condition remains **True** or until it becomes **True**. For more information, see [Do...Loop Statement (Visual Basic)](https://msdn.microsoft.com/en-us/library/eked04a7.aspx).

## [For Loops](javascript:void(0))

The **For**...**Next** construction performs the loop a set number of times. It uses a loop control variable, also called a counter, to keep track of the repetitions. You specify the starting and ending values for this counter, and you can optionally specify the amount by which it increases from one repetition to the next. For more information, see [For...Next Statement (Visual Basic)](https://msdn.microsoft.com/en-us/library/5z06z1kb.aspx).

## [For Each Loops](javascript:void(0))

The **For Each**...**Next** construction runs a set of statements once for each element in a collection. You specify the loop control variable, but you do not have to determine starting or ending values for it. For more information, see [For Each...Next Statement (Visual Basic)](https://msdn.microsoft.com/en-us/library/5ebk1751.aspx).

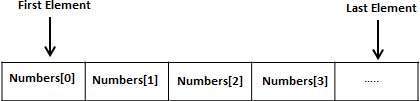
### Introduction to Arrays

By definition, an array is a list of variables with the same data type and name. 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 difficulty to declare 100 different names, 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)

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Creating Arrays in VB.Net

To declare an array in VB.Net, you use the Dim statement. For example,

Dim intData(30) ' an array of 31 elements

Dim strData(20) As String ' an array of 21 strings

Dim twoDarray(10, 20) As Integer 'a two dimensional array of integers

Dim ranges(10, 100) 'a two dimensional array

You can also initialize the array elements while declaring the array. For example,

Dim intData() As Integer = {12, 16, 20, 24, 28, 32}

Dim names() As String = {"Karthik", "Sandhya", \_

"Shivangi", "Ashwitha", "Somnath"}

Dim miscData() As Object = {"Hello World", 12d, 16ui, "A"c}

The elements in an array can be stored and accessed by using the index of the array. The following program demonstrates this:

Module arrayApl

Sub Main()

Dim n(10) As Integer ' n is an array of 11 integers '

Dim i, j As Integer

' initialize elements of array n '

For i = 0 To 10

n(i) = i + 100 ' set element at location i to i + 100

Next i

' output each array element's value '

For j = 0 To 10

Console.WriteLine("Element({0}) = {1}", j, n(j))

Next j

Console.ReadKey()

End Sub

End Module

When the above code is compiled and executed, it produces the following result:

Element(0) = 100

Element(1) = 101

Element(2) = 102

Element(3) = 103

Element(4) = 104

Element(5) = 105

Element(6) = 106

Element(7) = 107

Element(8) = 108

Element(9) = 109

Element(10) = 110

Dynamic Arrays

Dynamic arrays are arrays that can be dimensioned and re-dimensioned as par the need of the program. You can declare a dynamic array using the **ReDim** statement.

Syntax for ReDim statement:

ReDim [Preserve] arrayname(subscripts)

Where,

* The **Preserve** keyword helps to preserve the data in an existing array, when you resize it.
* **arrayname** is the name of the array to re-dimension.
* **subscripts** specifies the new dimension.

Module arrayApl

Sub Main()

Dim marks() As Integer

ReDim marks(2)

marks(0) = 85

marks(1) = 75

marks(2) = 90

ReDim Preserve marks(10)

marks(3) = 80

marks(4) = 76

marks(5) = 92

marks(6) = 99

marks(7) = 79

marks(8) = 75

For i = 0 To 10

Console.WriteLine(i & vbTab & marks(i))

Next i

Console.ReadKey()

End Sub

End Module

When the above code is compiled and executed, it produces the following result:

0 85

1 75

2 90

3 80

4 76

5 92

6 99

7 79

8 75

9 0

10 0

Multi-Dimensional Arrays

VB.Net allows multidimensional arrays. Multidimensional arrays are also called rectangular arrays.

You can declare a 2-dimensional array of strings as:

Dim twoDStringArray(10, 20) As String

or, a 3-dimensional array of Integer variables:

Dim threeDIntArray(10, 10, 10) As Integer

The following program demonstrates creating and using a 2-dimensional array:

Module arrayApl

Sub Main()

' an array with 5 rows and 2 columns

Dim a(,) As Integer = {{0, 0}, {1, 2}, {2, 4}, {3, 6}, {4, 8}}

Dim i, j As Integer

' output each array element's value '

For i = 0 To 4

For j = 0 To 1

Console.WriteLine("a[{0},{1}] = {2}", i, j, a(i, j))

Next j

Next i

Console.ReadKey()

End Sub

End Module

When the above code is compiled and executed, it produces the following result:

a[0,0]: 0

a[0,1]: 0

a[1,0]: 1

a[1,1]: 2

a[2,0]: 2

a[2,1]: 4

a[3,0]: 3

a[3,1]: 6

a[4,0]: 4

a[4,1]: 8

Jagged Array

A Jagged array is an array of arrays. The follwoing code shows declaring a jagged array named *scores* of Integers:

Dim scores As Integer()() = New Integer(5)(){}

The following example illustrates using a jagged array:

Module arrayApl

Sub Main()

'a jagged array of 5 array of integers

Dim a As Integer()() = New Integer(4)() {}

a(0) = New Integer() {0, 0}

a(1) = New Integer() {1, 2}

a(2) = New Integer() {2, 4}

a(3) = New Integer() {3, 6}

a(4) = New Integer() {4, 8}

Dim i, j As Integer

' output each array element's value

For i = 0 To 4

For j = 0 To 1

Console.WriteLine("a[{0},{1}] = {2}", i, j, a(i)(j))

Next j

Next i

Console.ReadKey()

End Sub

End Module

When the above code is compiled and executed, it produces the following result:

a[0][0]: 0

a[0][1]: 0

a[1][0]: 1

a[1][1]: 2

a[2][0]: 2

a[2][1]: 4

a[3][0]: 3

a[3][1]: 6

a[4][0]: 4

a[4][1]: 8

The Array Class

The Array class is the base class for all the arrays in VB.Net. It is defined in the System namespace. The Array class provides various properties and methods to work with arrays.

Properties of the Array Class

The following table provides some of the most commonly used **properties** of the **Array** class:

|  |  |
| --- | --- |
| **S.N** | **Property Name & Description** |
| 1 | **IsFixedSize**  Gets a value indicating whether the Array has a fixed size. |
| 2 | **IsReadOnly**  Gets a value indicating whether the Array is read-only. |
| 3 | **Length**  Gets a 32-bit integer that represents the total number of elements in all the dimensions of the Array. |
| 4 | **LongLength**  Gets a 64-bit integer that represents the total number of elements in all the dimensions of the Array. |
| 5 | **Rank**  Gets the rank (number of dimensions) of the Array. |

Methods of the Array Class

The following table provides some of the most commonly used **methods** of the **Array** class:

|  |  |
| --- | --- |
| **S.N** | **Method Name & Description** |
| 1 | **Public Shared Sub Clear (array As Array, index As Integer, length As Integer)**  Sets a range of elements in the Array to zero, to false, or to null, depending on the element type. |
| 2 | **Public Shared Sub Copy (sourceArray As Array, destinationArray As Array, length As Integer)**  Copies a range of elements from an Array starting at the first element and pastes them into another Array starting at the first element. The length is specified as a 32-bit integer. |
| 3 | **Public Sub CopyTo (array As Array, index As Integer)**  Copies all the elements of the current one-dimensional Array to the specified one-dimensional Array starting at the specified destination Array index. The index is specified as a 32-bit integer. |
| 4 | **Public Function GetLength (dimension As Integer) As Integer**  Gets a 32-bit integer that represents the number of elements in the specified dimension of the Array. |
| 5 | **Public Function GetLongLength (dimension As Integer) As Long**  Gets a 64-bit integer that represents the number of elements in the specified dimension of the Array. |
| 6 | **Public Function GetLowerBound (dimension As Integer) As Integer**  Gets the lower bound of the specified dimension in the Array. |
| 7 | **Public Function GetType As Type**  Gets the Type of the current instance (Inherited from Object). |
| 8 | **Public Function GetUpperBound (dimension As Integer) As Integer**  Gets the upper bound of the specified dimension in the Array. |
| 9 | **Public Function GetValue (index As Integer) As Object**  Gets the value at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer. |
| 10 | **Public Shared Function IndexOf (array As Array,value As Object) As Integer**  Searches for the specified object and returns the index of the first occurrence within the entire one-dimensional Array. |
| 11 | **Public Shared Sub Reverse (array As Array)**  Reverses the sequence of the elements in the entire one-dimensional Array. |
| 12 | **Public Sub SetValue (value As Object, index As Integer)**  Sets a value to the element at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer. |
| 13 | **Public Shared Sub Sort (array As Array)**  Sorts the elements in an entire one-dimensional Array using the IComparable implementation of each element of the Array. |
| 14 | **Public Overridable Function ToString As String**  Returns a string that represents the current object (Inherited from Object). |

For complete list of Array class properties and methods, please consult Microsoft documentation.

Example

The following program demonstrates use of some of the methods of the Array class:

Module arrayApl

Sub Main()

Dim list As Integer() = {34, 72, 13, 44, 25, 30, 10}

Dim temp As Integer() = list

Dim i As Integer

Console.Write("Original Array: ")

For Each i In list

Console.Write("{0} ", i)

Next i

Console.WriteLine()

' reverse the array

Array.Reverse(temp)

Console.Write("Reversed Array: ")

For Each i In temp

Console.Write("{0} ", i)

Next i

Console.WriteLine()

'sort the array

Array.Sort(list)

Console.Write("Sorted Array: ")

For Each i In list

Console.Write("{0} ", i)

Next i

Console.WriteLine()

Console.ReadKey()

End Sub

End Module

When the above code is compiled and executed, it produces the following result:

Original Array: 34 72 13 44 25 30 10

Reversed Array: 10 30 25 44 13 72 34

Sorted Array: 10 13 25 30 34 44 72

**Unit-4**

# Types of Procedures

We are going to do something **unconventional**.  Typically a programming language is **started** at the basics, like definition of variables and data types.  Other elements like functions/procedures come much later.  Here we are going to **begin** with **procedures.**  VB is developing into an important tool for designing the Graphical User Interface(GUI) in a window's application.  Event driven procedures  are necessary to achieve this.  Moreover we would also like to **start fooling with VB**.

Procedures are units/blocks of code that typically calculate values, manipulate text or control, or interact with database operations.  They are very useful for breaking down a large programs into smaller units.  Most often you will use a procedure to execute repeated or shared tasks to avoid code duplication - (Code duplication is not wrong merely inefficient).  You can also use them as building blocks that let you easily develop other related applications fast.

(from books on line)   
There are two major benefits of programming with procedures:

http://people.rit.edu/pnveme/Visual_Basic/Bullet1.gif  Procedures allow you to break your programs into discrete logical units, each of which you can debug more easily than an entire program without procedures.   
http://people.rit.edu/pnveme/Visual_Basic/Bullet1.gif  Procedures used in one program can act as building blocks for other programs, usually with little or no modification.

There are several types of procedures used in Visual Basic:

http://people.rit.edu/pnveme/Visual_Basic/Bullet2.gif[Sub](http://people.rit.edu/pnveme/Visual_Basic/Procedure_Types.html#Sub_proc) procedures do not return a value.   
http://people.rit.edu/pnveme/Visual_Basic/Bullet2.gif[Function](file:///C:\EMEM899\Function_Procedures.html) procedures return a value.   
http://people.rit.edu/pnveme/Visual_Basic/Bullet2.gifProperty procedures can return and assign values, and set references to objects.

## Sub Procedure

A Sub procedure is a block of code that is executed in response to an event. By breaking the code in a module into Sub procedures, it becomes much easier to find or modify the code in your application.

The syntax for a Sub procedure is: ( we will postpone the discussion on Private/Public/Static to later)

     [Private|Public][Static]Sub procedurename (arguments)

        statements

     End Sub

Each time the Sub procedure is called, the statements between Sub and End Sub are executed. Sub procedures can be placed in standard modules, class modules, and form modules. Sub procedures are by default Public in all modules, which means they can be called from anywhere in the application.

The arguments for a procedure are like a variable declaration, declaring values that are passed in from the calling procedure.  We will explore this after we have defined variables

In VB, it's useful to distinguish between two types of Sub procedures, event procedures and general procedures.

#### Event Procedures When an object in VB recognizes that an event has occurred, it automatically invokes the event procedure using the name corresponding to the event. Because the name establishes an association between the object and the code, event procedures are said to be attached to forms and controls.

Syntax for a control event

    Private Sub controlname\_eventname (arguments )   
        statements   
    End

Syntax for a form event

    Sub Private Sub Form\_eventname (arguments)   
            statements   
    End Sub

http://people.rit.edu/pnveme/Visual_Basic/Bullet3.gif  An event procedure for a control combines the control's actual name (specified in the Name property), an underscore (\_), and the event name. For instance, if you want a command button named cmdPlay to invoke an event procedure when it is clicked, use the procedure cmdPlay\_Click.

http://people.rit.edu/pnveme/Visual_Basic/Bullet3.gif  An event procedure for a form combines the word "Form" an underscore, and the event name. If you want a form to invoke an event procedure when it is clicked, use the procedure Form\_Click. (Like controls, forms do have unique names, but they are not used in the names of event procedures.) If you are using the MDI form, the event procedure combines the word "MDIForm," an underscore, and the event name, as in MDIForm\_Load.

**All event procedures use the same general syntax.**   
    
Although you can write event procedures from scratch, it's easier to use the code procedures provided by VB, which automatically include the correct procedure names. You can select a template in the Code Editor window by selecting an object from the Object box and then selecting a procedure from the Procedure box.

**We will now examine event procedures and how to program them**

http://people.rit.edu/pnveme/Visual_Basic/Bullet7.gif**Run the program we have created so far by clicking  the blue forward filled triangle on the toolbar**   
**The program compiles and executes.  You should see the image of the form with the controls we created.**

http://people.rit.edu/pnveme/Visual_Basic/Bullet7.gif**Click on any of the command button. Note that nothing happens !**

**While the command button by default can recognize the mouse click we have not instructed the program to respond to this event.**

http://people.rit.edu/pnveme/Visual_Basic/Bullet7.gif**Exit/terminate the program by clicking the " X " sign in the title bar of the application window**   
**This should put us back in VB.**

For the following I have retained default names for the various objects on the form.  You should work with the changes you had made earlier because this increases the effectiveness of what you learn today.

http://people.rit.edu/pnveme/Visual_Basic/Bullet4.gif  **Select Object View**   
**http://people.rit.edu/pnveme/Visual_Basic/Bullet4.gif  Select the second Command Button you created in the Properties Window**   
**http://people.rit.edu/pnveme/Visual_Basic/Bullet4.gif  Change the caption to read "Exit"**   
**http://people.rit.edu/pnveme/Visual_Basic/Bullet4.gif  Double click on the  second Command Button you created**

This should place the code view screen in the center area and provide a template for the click event procedure.  This is the mouse click event.  We are now ready to write our first event procedure

Private Sub Command2\_Click()   
    End  ' exits application - this is a comment - anything right of the single quote is ignored by compiler   
End Sub

http://people.rit.edu/pnveme/Visual_Basic/Bullet4.gif  **Run the application**   
**http://people.rit.edu/pnveme/Visual_Basic/Bullet4.gif  Click the command button**   
**The application should terminate and put us back into VB design environment**

We have successfully executed an event procedure.

### Event Driven Programming (EDP) :

The impetus for event driven programming comes from the popularity of the user interface(UI) that is now expected in all software.  Given the ease of navigation and the enhanced productivity that accompanies a software with the user interface, imagine the difficulty involved in getting the people to use UNIX.  Here you execute the programs on the command line.  Actually this is an exaggeration.  X-Windows and Motif have removed some of the pain, but they still lag significantly behind the PC windows or the Mac interface.

 We will examine some more aspects of event driven programing.  This time we will set up the first command button to accomplish something for us. We will select the button in a different way

http://people.rit.edu/pnveme/Visual_Basic/Bullet4.gif  **Select Code View**   
**http://people.rit.edu/pnveme/Visual_Basic/Bullet4.gif  Using the left drop down button at the top select the first Command Button you created**   
**http://people.rit.edu/pnveme/Visual_Basic/Bullet4.gif  Click on the right drop down button.  Examine the name of all the events that the object selected on the left supports.**   
**http://people.rit.edu/pnveme/Visual_Basic/Bullet4.gif  Lets place the following pieces of code by selecting the object and the event through the drop down list**

 Private Sub Command1\_Click()   
    Text1.Text = "command button 1 is clicked"     ' the Text property of Text1 is changes   
    Label1.Caption = "Why me ?"                ' the caption property of Label1 is changed   
End Sub

Private Sub Command2\_Click()   
    End   
End Sub

Private Sub Label1\_Click()   
    Label1.BackColor = vbGreen   ' the background color property of Label1 1 changes   
    
End Sub

Private Sub Text1\_LostFocus()   
    Label2.BorderStyle = 1       ' the border style property of Label2 is changed   
End Sub

http://people.rit.edu/pnveme/Visual_Basic/Bullet4.gif  **Run the application**

    When the application starts notice that the cursor is in the Textbox.  This is the same as saying that the Textbox has got the focus.

http://people.rit.edu/pnveme/Visual_Basic/Bullet4.gif  **Click the command button 1**

When you click on command1 - two events have simultaneously take place. (1) the Command Button 1 click event was launched and (2) the Textbox lost focus.  The click event of the command button changed the text in the Textbox and the caption of Label1.  The loss of focus of the Textbox caused a change in the appearance of Label2.  Clicking Label1  ( generally not done - since label basically is a container for descriptive text) causes Label1 to change color - here a VB constant "vbGreen" is used .

 http://people.rit.edu/pnveme/Visual_Basic/Bullet4.gif  **Click the command button 2**

The application should terminate and put us back into VB design environment

## General Procedures :

We saw the event procedure respond to an event triggered by the mouse or the cursor.  A general procedure is used to   accomplish a specific task.  It must be deliberately invoked by the program.   By contrast, an event procedure remains idle until called upon to respond to events caused by the user or triggered by the system.

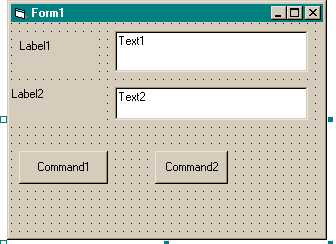
One reason to write a general procedure is that is that several different event procedures might need the same actions performed. A good programming strategy is to put common statements in a separate procedure (a general procedure) and have your event procedures call it.

Avoid writing event procedures until you have frozen the **Name** of the **Object**.  This allows you to use event procedure templates provided by VB.  If the name is changed after writing the procedures then you will have to change the name of the procedure too for VB to recognize it as an event procedure.  If VB does not recognize it as an event procedure it is considered a general procedure.

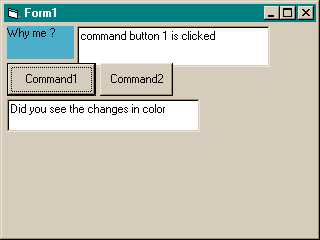
It is a good idea to have event procedures call general procedures

For the code below:

### Screen Shot of the Form at design Time



### Screen shot of the form at Run Time



### Examples of General Procedures

Private Sub Command1\_Click()   
         Text1.Text = "command button 1 is clicked"     ' the Text property of Text1 is changes   
         Label1.Caption = "Why me ?"  ' the caption property of Label1 is changed   
    
         ' Example of a general procedure   
         Label\_ColorChange    ' a general procedure without arguments   
         Call Locate\_Text       ' a similar one but notice another way to call it   
End Sub   
    
Private Sub Command2\_Click()   
   MsgBox "Do You want to quit", vbOKOnly, "Quit" ' Using built in MsgBox Function   
   End   
End Sub   
Private Sub Label1\_Click()   
    Label1.BackColor = vbGreen       ' the background color property of Label1 1 changes   
End Sub

Private Sub Text1\_LostFocus()   
     Label2.BorderStyle = 1           ' the border style property of Label2 is changed   
End Sub   
Private Sub Label\_ColorChange() ' a general procedure   
    Label1.BackColor = RGB(75, 175, 203)    'Using RGB function   
    Label2.BackColor = RGB(25, 100, 100)   
    Label2.ForeColor = RGB(255, 255, 255)   
    Text2.Text = "Did you see the changes in color"   
End Sub   
Private Sub Locate\_Text()    'another general procedure   
    Label1.Left = 50            ' location the objects in code   
    Label1.Top = 50   
    Label1.Height = 500   
    Label1.Width = 1000   
    
    With Label2     ' the built in with keyword simplifies coding   
        .Left = 50   
        .Top = Label1.Top + Label1.Height + 50   
        .Height = Label1.Height   
        .Width = Label1.Width   
    End With   
    
    Text1.Left = Label1.Left + Label1.Width + 50   
    Text1.Top = Label1.Top   
    
    With Command1   
        .Left = 50   
        .Top = Label2.Top + Label2.Height + 50   
    End With   
    
    Command2.Top = Command1.Top   
    Command2.Left = Command1.Left + Command1.Width + 50   
    Command2.Height = Command1.Height   
    
    Text2.Left = 50   
    Text2.Top = Command1.Top + Command1.Height + 50   
    
End Sub

**We are now familiar with both kinds of Sub procedures**

## Function Procedures

 A Function procedure is another kind of  procedure, similar to a Sub procedure  for it  can take arguments, perform a series of statements, and change the value of its arguments. However, unlike a Sub procedure, a Function procedure can return a value to the calling procedure.

There are three differences between Sub and Function procedures: (books on line)

http://people.rit.edu/pnveme/Visual_Basic/Bullet3.gif  Generally, you call a function by including the function procedure name and arguments on the right side of a larger statement or expression   
        returnvalue = function()

http://people.rit.edu/pnveme/Visual_Basic/Bullet3.gif  Function procedures have data types, just as variables do. This determines the type of the return value. (In the absence of an As clause, the type is the default Variant type.)

http://people.rit.edu/pnveme/Visual_Basic/Bullet3.gif  You return a value by assigning it to the procedurename itself. When the Function procedure returns a value, this value can then become part of a larger expression.

Visual Basic includes built-in, or intrinsic functions, like Sqr, Cos or Chr. In addition, you can use the Function statement to write your own Function procedures.

The syntax for a Function procedure is:

[Private|Public][Static]Function procedurename (arguments) [As type]

    statements

End Function

## Property Procedures

**Property procedures are typically use to set an object's property or to get the value of an object's property.  This will be useful when we define our own classes and objects.**

**Here we  just accept the following information**

Visual Basic provides three kinds of property procedures depending on the purpose

Property Get Returns the value of a property.

Property Let Sets the value of a property.

Property Set Sets the value of an object property (that is, a property that contains a reference to an object).

Each of these property procedures has a particular role to play in defining a property. The typical property will be made up of a pair of property procedures: A Property Get to retrieve the property value, and a Property Let or Property Set to assign a new value.

|  |
| --- |
| **Passing Arguments** |
|  | |

|  |
| --- |
| **Using Global Variables** |
|  |

|  |  |
| --- | --- |
|  | In the previous lesson, we saw that you could declare a global variable outside of any procedure. When using various procedures in a code file, one of the characteristics of a global variable is that it is automatically accessible to other procedures. Still, a global variable can use access modifiers that would control its access: |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| * **Private**: A private global variable can be accessed  by any procedure of the same module. No procedure of another module, even of the same program, can access it   Global Variables * **Friend**: A friendly global variable can be accessed by any procedure of any module of the same project. A procedure of another program cannot access that variable   Friend Access Modifier * **Public**: A public global variable can be accessed by any procedure of its project and procedures of other projects   Access Levels   Based on this characteristic of the procedures of a module having access to global variables of the same program, you can declare such variables and initialize or modify them in any procedure of the same code file.   |  | | --- | | **http://www.functionx.com/images/practical1.gifPractical Learning: Using Global Variables** | |  |  1. To use global variables, change the document as follows: 2. Module Geometry 3. Private Length As Double 4. Private Width As Double 5. Private Sub GetLength() 6. Length = InputBox(&quot;Enter Rectangle Length:&quot;) 7. End Sub 8. Private Sub GetWidth() 9. Width = InputBox(&quot;Enter Rectangle Width:&quot;) 10. End Sub 11. Private Function CalculatePerimeter() As Double 12. CalculatePerimeter = (Length + Width) \* 2 13. End Function 14. Public Function Main() As Integer 15. Dim Perimeter As Double 16. GetLength() 17. GetWidth() 18. Perimeter = CalculatePerimeter() 19. MsgBox("=-= Square Characteristics=-=" & vbCrLf & 20. "Length: " & vbTab & Length & vbCrLf & 21. "Width: " & vbTab & Width & vbCrLf & 22. "Perimeter: " & Perimeter) 23. Return 0 24. End Function   End Module   1. To execute the program, on the Standard toolbar, click the Start Debugging button Start Debugging 2. Enter the length as **32.08** and the with as **24.84** 3. Close the message box and return to your programming environment  |  | | --- | | **Introduction to Arguments** | |  |   So far, to use a value in a procedure, we had to declare it. In some cases, a procedure may need an external value in order to carry its assignment. A value that is supplied to a procedure is called an argument.  When creating a procedure that will use an external value, declare the argument that represents that value between the parentheses of the procedure. For a procedure, the syntax you use would be:  Sub *ProcedureName*(*Argument*)    End Sub  If you are creating a function, the syntax would be:  Function *ProcedureName*(*Argument*) As *DataType*    Function Sub  The argument must be declared as a normal variable, omitting the **Dim** keyword. Here is an example that creates a function that takes a string as argument:  Function CalculatePayroll(strName As String) As Double    Function Sub  A certain procedure can take more than one argument. In this case, in the parentheses of the procedure, separate the arguments with a comma. Here is an example of a procedure that takes two arguments:  Sub EvaluateInvoice(EmplName As String, HourlySalary As Currency)    End Sub  In the body of a procedure that takes one or more arguments, use the argument(s) as you see fit as if they were locally declared variables. For example, you can involve them with values inside of the procedure. You can also exclusively use the values of the arguments to perform the assignment.   |  | | --- | | **Calling a Procedure With Argument** | |  |   To call a procedure that takes an argument, type its name and a space, followed by a value for each argument between parentheses. The value provided for an argument is also called a parameter. If there is more than one argument, separate them with a comma. Here is an example:  Module Exercise  Private Function GetFullName$(strFirst As String,  strLast As String)  Dim FName As String  FName = strFirst & " " & strLast  GetFullName = FName  End Function  Public Function Main() As Integer  Dim FirstName, LastName As String  Dim FullName As String  Dim ComputerLanguage As String = "Visual Basic"  FirstName = inputbox("Enter First Name: ")  LastName = inputbox("Enter Last Name: ")  FullName = GetFullName(FirstName, LastName)  msgbox("Hello, " & FullName)  Welcome(ComputerLanguage)  Return 0  End Function  Sub Welcome(ByVal strLanguage As String)  msgbox("Welcome to the wonderful world of " & strLanguage)  End Sub  End Module  As mentioned previously, you can also use the **Call** keyword to call a procedure.  When you call a procedure that takes more than one argument, you must provide the values of the arguments in the exact order they are listed inside of the parentheses of the function. Fortunately, you don't have to. If you know the names of the arguments, you can type them in any order and provide a value for each. To do that, on the right side of each argument, type the := operator followed by the desired value for the argument. Here are examples:  Public Module Exercise  Private Function GetFullName$(MI As String,  LastName As String,  FirstName As String)  GetFullName = FirstName & " " & MI & " " & LastName  End Function  Public Function Main() As Integer  Dim FullName As String  Dim ComputerLanguage As String = "VBasic"  FullName = GetFullName(LastName:="Roberts", FirstName:="Alan", MI:="R.")  MsgBox("Hello " & FullName)  Call Welcome(ComputerLanguage)  Return 0  End Function  Private Sub Welcome(ByVal strLanguage As String)  MsgBox("Welcome to the wonderful world of " & strLanguage)  End Sub  End Module   |  | | --- | | **http://www.functionx.com/images/practical1.gifPractical Learning: Passing Arguments to a Procedure** | |  |  1. To pass arguments to a function, change the file as follows (when you type the argument, Microsoft Visual Studio, actually the Visual Basic language parser, will add the **ByVal** keywords; in the next sections, we will learn what that keyword means; for now, keep it but ignore it): 2. Module Geometry 3. Private Function GetValue(TypeOfValue As String) As Double 4. Dim Value As Double 5. Value = InputBox("Enter the " & TypeOfValue & ":") 6. Return Value 7. End Function 8. Private Function CalculatePerimeter(ByVal Length As Double, 9. ByVal Width As Double) As Double 10. CalculatePerimeter = (Length + Width) \* 2 11. End Function 12. Public Function Main() As Integer 13. Dim L As Double, W As Double 14. Dim Perimeter As Double 15. L = GetValue("Length") 16. W = GetValue("Width") 17. Perimeter = CalculatePerimeter(L, W) 18. MsgBox("=-= Square Characteristics=-=" & vbCrLf & 19. "Length: " & L & vbCrLf & 20. "Width: " & W & vbCrLf & 21. "Perimeter: " & Perimeter) 22. Return 0 23. End Function   End Module   1. To execute the program, on the main menu, click Debug -> Start Debugging 2. Enter the length as **44.14** and the with as **30.76** 3. Close the message box and return to your programming environment |  |

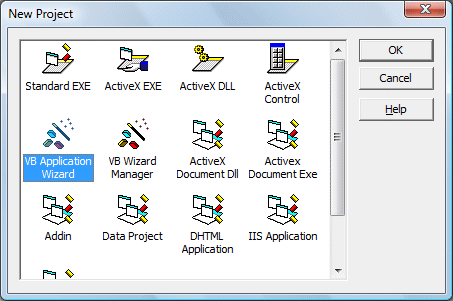
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | **Techniques of Passing Arguments** | |  |      |  | | --- | | **Passing an Argument By Value** | |  |   When calling a procedure that takes an argument, we were supplying a value for that argument. When this is done, the procedure that is called makes a copy of the value of the argument and makes that copy available to the calling procedure. That way, the argument itself is not accessed. This is referred to as passing an argument by value. This can be reinforced by typing the **ByVal** keyword on the left side of the argument. Here is an example:  Private Sub Welcome(ByVal strLanguage As String)  MsgBox("Welcome to the wonderful world of " & strLanguage)  End Sub  If you create a procedure that takes an argument by value and you have used the **ByVal** keyword on the argument, when calling the procedure, you don't need to use the **ByVal** keyword; just the name of the argument is enough, as done in the examples on arguments so far. Here is an example:  Public Module Exercise  Public Function Main() As Integer  Dim ComputerLanguage As String = "Visual Basic"  Welcome(ComputerLanguage)  Return 0  End Function  Private Sub Welcome(ByVal strLanguage As String)  MsgBox("Welcome to the wonderful world of " & strLanguage)  End Sub  End Module  This would produce:  Arguments   |  | | --- | | **Passing an Argument By Reference** | |  |   An alternative to passing an argument as done so far is to pass the address of the argument to the procedure. When this is done, the procedure doesn't receive a simple copy of the value of the argument: the argument is accessed by its address. That is, at its memory address. With this technique, any action carried on the argument will be kept. If the value of the argument is modified, the argument would now have the new value, dismissing or losing the original value it had. This technique is referred to as passing an argument by reference. Consider the following program:  Public Module Exercise  Private Function Addition#(ByVal Value1 As Double, ByVal Value2 As Double)  Value1 = InputBox("Enter First Number: ")  Value2 = InputBox("Enter Second Number: ")  Addition = Value1 + Value2  End Function  Public Function Main() As Integer  Dim Result As String  Dim Number1, Number2 As Double  Result = Addition(Number1, Number2)  MsgBox(Number1 & " + " & Number2 & " = " & Result)  Return 0  End Function  End Module  Here is an example of running the program:  http://www.functionx.com/visualbasic/functions/inputboxes/byval1.gif  http://www.functionx.com/visualbasic/functions/inputboxes/byval2.gif  http://www.functionx.com/visualbasic/functions/msgboxes/byval1.gif  Notice that, although the values of the arguments were changed in the Addition() procedure, at the end of the procedure, they lose the value they got in the function. If you want a procedure to change the value of an argument, you can pass the argument by reference.  To pass an argument by reference, on its left, type the **ByRef** keyword. This is done only when defining the procedure. When the procedure finishes with the argument, the argument would keep whatever modification was made on its value. Now consider the same program as above but with arguments passed by reference:  Public Module Exercise  Private Function Addition#(ByRef Value1 As Double, ByRef Value2 As Double)  Value1 = InputBox("Enter First Number: ")  Value2 = InputBox("Enter Second Number: ")  Addition = Value1 + Value2  End Function  Public Function Main() As Integer  Dim Result As String  Dim Number1, Number2 As Double  Result = Addition(Number1, Number2)  MsgBox(Number1 & " + " & Number2 & " = " & Result)  Return 0  End Function  End Module  Here is an example of running the program:  http://www.functionx.com/visualbasic/functions/inputboxes/byval1.gif  http://www.functionx.com/visualbasic/functions/inputboxes/byval2.gif  http://www.functionx.com/visualbasic/msgboxes/byref1.gif  Using this technique, you can pass as many arguments by reference and as many arguments by value as you want. As you may guess already, this technique is also used to make a procedure return a value, which a regular procedure cannot do. Furthermore, passing arguments by reference allows a procedure to return as many values as possible while a regular function can return only one value.   |  | | --- | | **http://www.functionx.com/images/practical1.gifPractical Learning: Passing Arguments by Reference** | |  |  1. To pass an argument by reference, change the file as follows: 2. Module Geometry 3. Private Sub GetValues(ByRef Length As Double, ByRef Width As Double) 4. Length = InputBox("Enter the length:") 5. Width = InputBox("Enter the width:") 6. End Sub 7. Private Function CalculatePerimeter(ByVal Length As Double, 8. ByVal Width As Double) As Double 9. CalculatePerimeter = (Length + Width) \* 2 10. End Function 11. Private Function CalculateArea(ByVal Length As Double, \_ 12. ByVal Width As Double) As Double 13. CalculateArea = Length \* Width 14. End Function 15. Private Sub ShowCharacteristics(ByVal Length As Double, 16. ByVal Width As Double) 17. Dim Result As String 18. Result = "=-= Rectangle Characteristics =-=" & vbCrLf & 19. "Length: " & vbTab & vbTab & CStr(Length) & vbCrLf & 20. "Width: " & vbTab & vbTab & CStr(Width) & vbCrLf & 21. "Perimeter: " & vbTab & 22. CalculatePerimeter(Length, Width) & vbCrLf & 23. "Area: " & vbTab & vbTab & 24. CalculateArea(Length, Width) 25. MsgBox(Result) 26. End Sub 27. Public Function Main() As Integer 28. Dim L As Double, W As Double 29. GetValues(L, W) 30. ShowCharacteristics(L, W) 31. Return 0 32. End Function   End Module   1. To execute the program, on the Standard toolbar, click the Start Debugging button Start Debugging 2. Enter the length as **24.55** and the width as **22.85**   http://www.functionx.com/visualbasic/msgboxes/byref2.gif 3. Close the message box and return to your programming environment  |  | | --- | | **Other Techniques of Passing Arguments** | |  |      |  | | --- | | **Optional Arguments** | |  |   If you create a procedure that takes one or more arguments, whenever you call that procedure, you must provide a value for the argument(s). Otherwise,, you would receive an error. If such an argument is passed with the same value over and over again, you may be tempted to remove the argument altogether. In some cases, although a certain argument is passed with the same value most of the time, you still have situations in which you want the user to decide whether to pass a value or not for the argument, you can declare the value optional. In other words, you can create the argument with a default value so that the user can call the procedure without passing a value for the argument, thus passing a value only when necessary. Such an argument is called default or optional.  Imagine you write a procedure that will be used to calculate the final price of an item after discount. The procedure would need the discount rate in order to perform the calculation. Such a procedure could look like this:  Function CalculateNetPrice#(ByVal DiscountRate As Double)  Dim OrigPrice#  OrigPrice = InputBox("Please enter the original price:")  Return OrigPrice - (OrigPrice \* DiscountRate / 100)  End Function  Since this procedure expects an argument, if you do not supply it, the following program would not compile:  Public Module Exercise  Function CalculateNetPrice#(ByVal DiscountRate As Double)  Dim OrigPrice#  OrigPrice = InputBox("Please enter the original price:")  Return OrigPrice - (OrigPrice \* DiscountRate / 100)  End Function  Public Function Main() As Integer  Dim FinalPrice#  Dim Discount# = 15 ' That is 25% = 25  FinalPrice = CalculateNetPrice(Discount)  MsgBox("Final Price = " & FinalPrice)  Return 0  End Function  End Module  Here is an example of running the program:  Optional Argument  Optional Argument  Most of the time, a procedure such as ours would use the same discount rate over and over again. Therefore, instead of supplying an argument all the time, you can define an argument whose value would be used whenever the function is not provided with the argument.  To specify that an argument is optional, when creating its procedure, type the **Optional** keyword to the left of the argument's name and assign it the default value. Here is an example:  Public Module Exercise  Function CalculateNetPrice#(Optional ByVal DiscountRate As Double = 20)  Dim OrigPrice#  OrigPrice = InputBox("Please enter the original price:")  Return OrigPrice - (OrigPrice \* DiscountRate / 100)  End Function  Public Function Main() As Integer  Dim FinalPrice#  Dim Discount# = 15 ' That is 25% = 25  FinalPrice = CalculateNetPrice()  MsgBox("Final Price = " & FinalPrice)  Return 0  End Function  End Module  Here is an example of running the program:  Optional Argument  Optional Argument  If a procedure takes more than one argument, you can provide a default argument for each and select which ones would have default values. If you want all arguments to have default values, when defining the procedure , provide the **Optional** keyword for each and assign it the desired default value. Here is an example:  Public Module Exercise  Function CalculateNetPrice#(Optional ByVal Tax As Double = 5.75,  Optional ByVal Discount As Double = 25,  Optional ByVal OrigPrice As Double = 245.55)  Dim DiscountValue As Double = OrigPrice \* Discount / 100  Dim TaxValue As Double = Tax / 100  Dim NetPrice As Double = OrigPrice - DiscountValue + TaxValue  Dim Result As String  Result = "Original Price: " & vbTab & CStr(OrigPrice) & vbCrLf &  "Discount Rate: " & vbTab & CStr(Discount) & "%" & vbCrLf &  "Tax Amount: " & vbTab & CStr(Tax)  MsgBox(Result)  Return NetPrice  End Function  Public Function Main() As Integer  Dim FinalPrice As Double  FinalPrice = CalculateNetPrice()  MsgBox("Final Price: " & CStr(FinalPrice))  Return 0  End Function  End Module  This would produce:  Optional Arguments  Optional Arguments  If a procedure takes more than one argument as above, remember that some arguments can be specified as optional. In this case, when calling the procedure, any argument that does not have a default value must be passed with a value. When creating a procedure that takes more than one argument, the argument(s) that has(have) default value(s) must be the last in the procedure. This means that:   * If a procedure takes two arguments and one argument has a default value, this optional argument must be the second * If a procedure is taking three or more arguments and two or more arguments have default values, these optional arguments must by placed to the right of the non-optional argument(s).   Because of this, when calling any procedure in the Visual Basic language, you must know what, if any, argument is optional and which one is not.  If a procedure takes two arguments and one argument has a default value, when calling this procedure, you can pass only one value. In this case, the passed value would be applied on the first argument. If a procedure takes more than two arguments and two or more arguments have a default value, when calling this procedure, you can provide only the value(s) of the argument that is (are) not optional. If you want to provide the value of one of the arguments but that argument is not the first optional, you can leave empty the position(s) of the other argument(s) but remember to type a comma to indicate that the position is that of an argument that has a default value. Here is an example:  Public Module Exercise  Function CalculateNetPrice(ByVal AcquiredPrice As Double,  ByVal MarkedPrice As Double,  Optional ByVal TaxRate As Double = 5.75,  Optional ByVal DiscountRate As Double = 25) As Double  Dim DiscountAmount As Double = MarkedPrice \* DiscountRate / 100  Dim TaxAmount As Double = MarkedPrice \* TaxRate / 100  Dim NetPrice As Double = MarkedPrice - DiscountAmount + TaxAmount  Dim Result As String  Result = "Price Acquired: " & vbTab & CStr(AcquiredPrice) & vbCrLf &  "Marked Price: " & vbTab & CStr(MarkedPrice) & vbCrLf &  "Discount Rate: " & vbTab & CStr(DiscountRate) & "%" & vbCrLf &  "Discount Amt: " & vbTab & CStr(DiscountAmount) & vbCrLf &  "Tax Rate: " & vbTab & CStr(TaxRate) & "%" & vbCrLf &  "Tax Amount: " & vbTab & CStr(TaxAmount)  MsgBox(Result)  Return NetPrice  End Function  Public Function Main() As Integer  Dim FinalPrice As Double  FinalPrice = CalculateNetPrice(225.55, 150.55, , 40)  MsgBox("Final Price: " & CStr(FinalPrice))  Return 0  End Function  End Module  This would produce:  Optional Arguments  Optional Arguments     |  | | --- | | **Procedure Overloading** | |  |   A program involves a great deal of names that represent variables and procedures of various kinds. The compiler does not allow two variables to have the same name in the same procedure (or in the same scope). Although two procedures should have unique names in the same program, you are allowed to use the same name for different procedures of the same program following certain rules.  The ability to have various procedures with the same name in the same program is referred to as overloading. The most important rule about procedure overloading is to make sure that each one of these procedures has a different number or different type(s) of arguments.   |  | | --- | | **Practical LearningPractical Learning: Overloading a Procedure** | |  |   The moment of inertia is the ability of a beam to resist bending. It is calculated with regard to the cross section of the beam. Because it depends on the type of section of the beam, its calculation also depends on the type of section of the beam. In this exercise, we will review different formulas used to calculate the moment of inertia. Since this exercise is for demonstration purposes, you do not need to be a Science Engineering major to understand it.  The Moment Of Inertia   1. To start a new project, on the main menu, click File -> New Project... 2. In the middle list, click Console Application 3. Change the Name to **MomentOfInertia1** 4. Click OK 5. In the Solution Explorer, right-click Module1.vb and click Rename 6. Type **MomentOfInertia.vb** and press Enter 7. To calculate the moment of inertia of a rectangle, change the file as follows: 8. Module MomentOfInertia 9. ' Moment of Inertia 10. ' Rectangle 11. Private Function MomentOfInertia(ByVal b As Double, 12. ByVal h As Double) As Double 13. Return b \* h \* h \* h / 3 14. End Function 15. Public Sub Main() 16. Dim Base As Double, Height As Double 17. Base = InputBox("Enter the base of the Rectangle") 18. Height = InputBox("Enter the height of the Rectangle") 19. MsgBox("Moment of inertia with regard to the X axis" & vbCrLf & 20. "I = " & CStr(MomentOfInertia(Base, Height)) & "mm") 21. End Sub   End Module   1. To execute the program, on the main menu, click Debug -> Start Debugging 2. Enter the base as **3.25**   Overloading a Procedure 3. Enter the height as **2.85**   Overloading a Procedure   Overloading a Procedure 4. Close the DOS window and return to your programming environment 5. Here are the formulas to calculate the moment of inertia for a semi-circle:  |  | | --- | | The Moment of Inertia for a Circle  A circle, and thus a semi-circle, requires only a radius. Since the other version of the MomentOfInertia() function requires two arguments, we can overload it by providing only one argument, the radius. |  1. To calculate the moment of inertia of a rectangle, change the file as follows:  |  | | --- | | Module MomentOfInertia  ' Moment of Inertia  ' Rectangle  Private Function MomentOfInertia(ByVal b As Double,  ByVal h As Double) As Double  Return b \* h \* h \* h / 3  End Function  ' Semi-Circle  Function MomentOfInertia(ByVal R As Double) As Double  Const PI As Double = 3.14159  Return R \* R \* R \* R \* PI / 8  End Function  Public Sub Main()  ' Dim Base As Double, Height As Double  Dim Radius As Double  ' Base = InputBox("Enter the base of the Rectangle:")  ' Height = InputBox("Enter the height of the Rectangle")  ' MsgBox("Moment of inertia with regard to the X axis" & vbCrLf & \_  ' "I = " & CStr(MomentOfInertia(Base, Height)) & "mm")  Radius = InputBox("Enter the radius of the semi-circle:")  MsgBox("Moment of inertia of a semi-circle with " &  "regard to the X axis:" & vbCrLf &  "I = " & CStr(MomentOfInertia(Radius)) & "mm")  End Sub  End Module |  1. To execute the program, on the Standard toolbar, click the Start Debugging button Start Debugging 2. Enter the radius as **6.35**   http://www.functionx.com/visualbasic/inputboxes/overloading3.gif   http://www.functionx.com/visualbasic/msgboxes/overloading3.gif 3. Close the DOS window and return to your programming environment 4. On the main menu, click File -> Close Solution (Microsoft Visual Studio) or File -> Close Project (Microsoft Visual Basic 2010 Express) 5. When asked whether you want to save, click Discard |

# Creating Menus for Your Applications

Menu bar is the standard feature of most windows applications. The main purpose of the menus is for easy navigation and control of an application. Some of the most common menu items are File, Edit, View, Tools, Help and more. Each item on the main menu bar also provide a list of options or  in the form of a pull-down menu. When you create a Visual Basic 6 program, you need not include as many menu items as a full fledge Windows application such as Microsoft Words. What you need is to include those menu items that can improve the ease of using your program by the user, and not to confuse the user with unnecessary items. Adding menu bar is relatively easy  to accomplish in Visual Basic. There are two ways to add menus to your application, one way is to use the Visual Basic's Application Wizard and the other way is to use the menu editor.

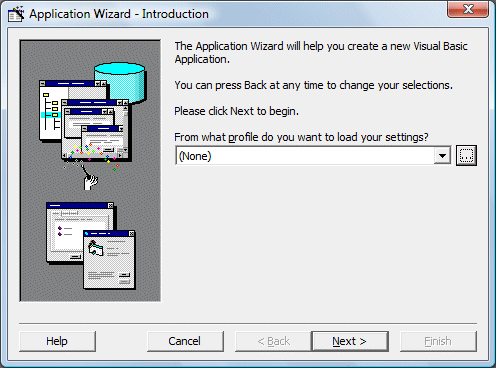
### Adding Menu Bar Using Visual Basic's Application Wizard

The easiest way to add menu bar to your application is by using Visual Basic's Application Wizard. This wizard allows the user to insert fully customized  standard windows menus into his or her application. To start using Visual Basic's Application Wizard, you click on the Application Wizard icon at the Visual Basic new project dialog box, as shown in Figure 37.1 below:



##### Figure: New Project Window

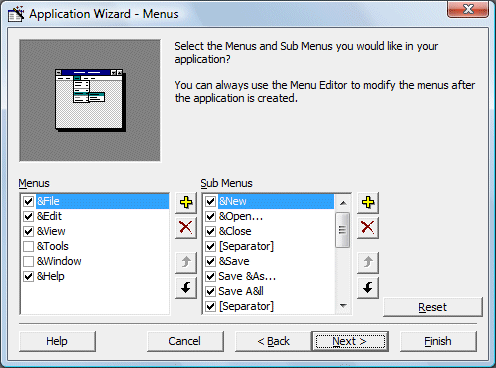
When you click on the VB Application wizard, the introduction dialog box will appear, as shown in Figure 37.2. As you are not loading any default setting, just click on the Next button.

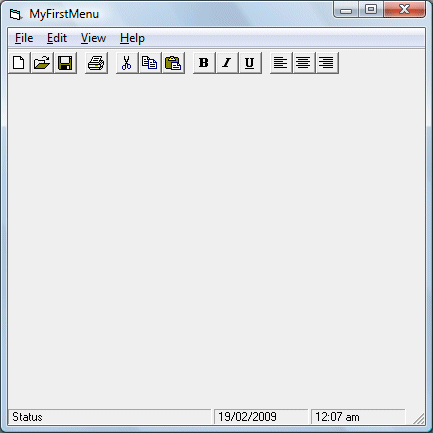


##### Figure

After clicking the Next button, the interface type dialog box will be displayed, as shown in There are three choices of interface available for your project. As we currently not creating a Multiple Document Interface (MDI), we choose Single Document Interface (SDI). You can also type the project name in the textbox below, here I am using MyFirstMenu.

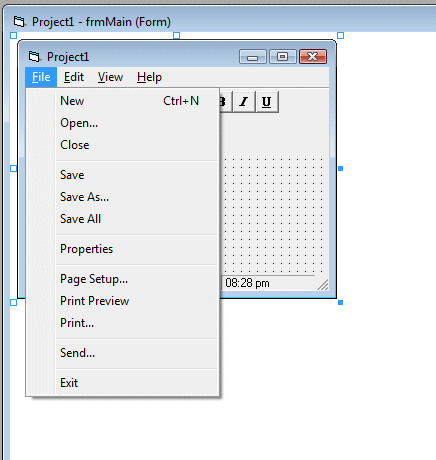
Clicking the Next button wiill bring up a list of menus and submenus that you can add them to your application. Check to select a menu item and uncheck to unselect a menu item as shown in Figure 37.4. Let say we choose all the menus and click next, then you will get an interface comprises File, Edit, View and Help menus, as shown in Figure 37.5





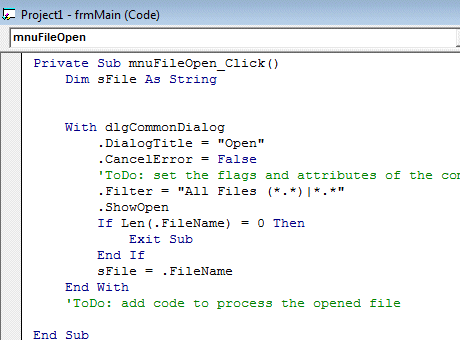
##### Figure

you click on any menu item, a list of drop-down submenu items will be displayed. For example, if you click on the File menu, the list of submenu items such as New, Open, Save, Save As and more will be displayed, as shown in Figure 37.6



##### Figure

Clicking on any of the dropped down menu item will show the code associated with it, and this is where you can modify the code to suit your programming needs. For example, clicking on the item Open will reveal the following code:



##### Figure

Now, I will show you how to modify the code in order to open a graphic file and display it in an image box. For this program, you have to insert a Image box into the form. Next add the following lines so that the user can open graphic files of different formats.

.Filter = "Bitmaps(\*.BMP)|\*.BMP|Metafiles(\*.WMF)|\*. WMF|Jpeg Files(\*.jpg)|\*.jpg|GIF Files(\*.gif)|\*.gif|Icon Files(\*.ico)|\*.ico|All Files(\*.\*)|\*.\*".

Then, you need to load the image into the Image box with the following code:

Image1.Picture = LoadPicture(.FileName)

Also set the Stretch property of the Image box to true so that the image loaded can resize by itself. Please note that each menu item is a special control, so it has a name too. The name for the menu File in this example is mnuFileOpen.

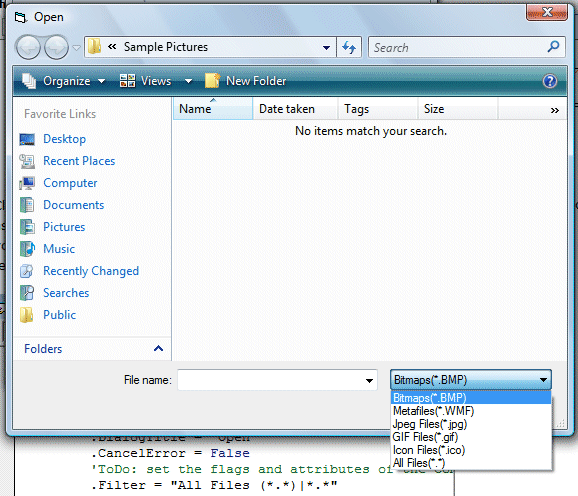
The complete code is as follows:

Private Sub mnuFileOpen\_Click()  
Dim sFile As String  
  
  
With dlgCommonDialog  
.DialogTitle = "Open"  
.CancelError = False

'ToDo: set the flags and attributes of the common dialog control  
.Filter = "Bitmaps(\*.BMP)|\*.BMP|Metafiles(\*.WMF)|\*. WMF|Jpeg Files(\*.jpg)|\*.jpg|GIF Files(\*.gif)|\*.gif|Icon Files(\*.ico)|\*.ico|All Files(\*.\*)|\*.\*"

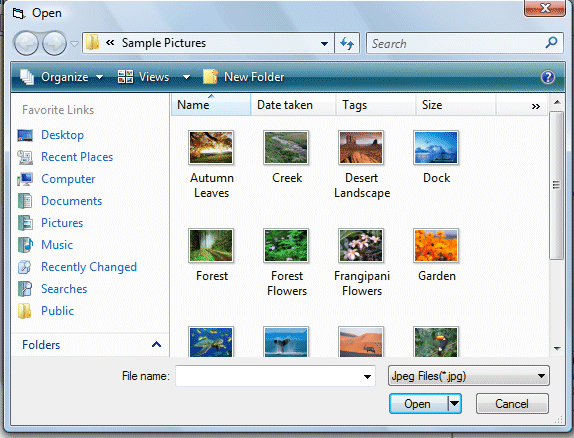
.ShowOpen  
Image1.Picture = LoadPicture(.FileName)  
  
If Len(.FileName) = 0 Then  
Exit Sub  
End If  
sFile = .FileName  
End With  
'ToDo: add code to process the opened file  
  
End Sub

When you run the program and click on the File menu and then the submenu Open, the following Open dialog box will be displayed, where you can look for graphic files of various formats to load it into the image box.



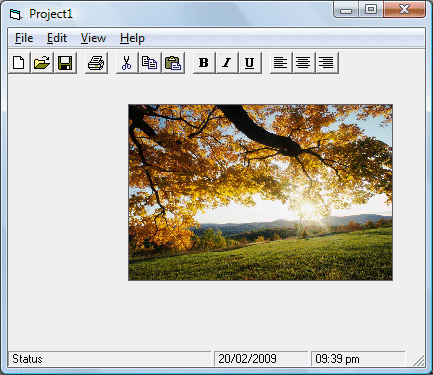
##### Figure

For example, selecting the jpeg file will allow you to choose the images of jpeg format,



##### Figure

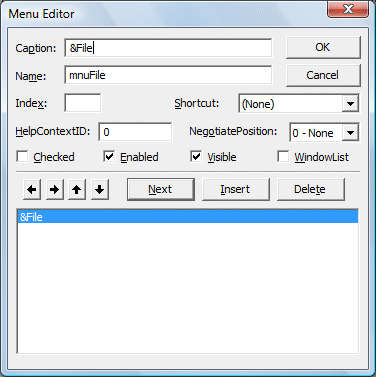
Clicking on the particular picture will load it into the image box,



##### Figure

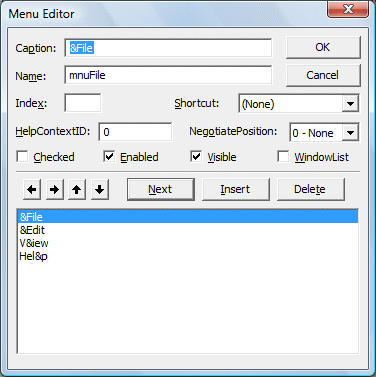
### Adding Menu Bar Using Menu Editor

To start adding menu items to your application, open an existing project or start a new project, then click on Tools in the menu bar of the Visual Basic IDE and select Menu Editor. When you click on the Menu Editor, the Menu Editor dialog will appear. In the Menu Editor dialog , key in the first item File in the caption text box. You can use the ampersand ( & ) sign in front of F so that F will be underlined when it appears in the menu, and F will become the hot key to initiate the action under this item by pressing the Alt key and the letter F. After typing &File in the Caption text box, move to the name textbox to enter the name for this menu item, you can type in mnuFile here. Now, click the Next button and the menu item &File will move into the empty space below, as shown in Figure 37.11:



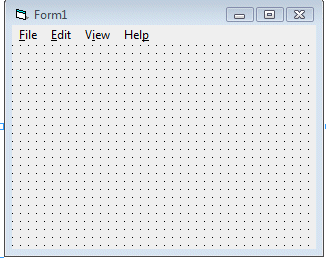
##### Figure 37.11

You can then add in other menu items on the menu bar by following the same procedure,



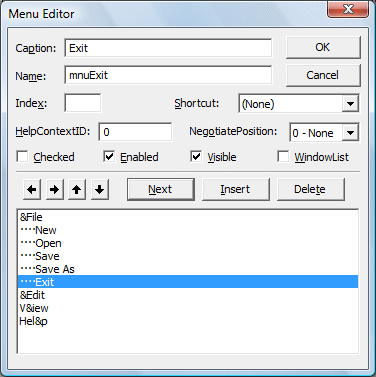
##### Figure

when you click Ok, the menu items will be shown on the menu bar of the form.



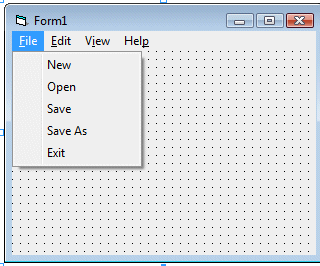
##### Figure

Now, you may proceed to add the sub menus. In the Menu Editor, click on the Insert button between File and Exit and then click the right arrow key, and the dotted line will appear. This shows the second level of the menu, or the submenu. Now key in the caption and the name. Repeat the same procedure to add other submenu items. Here, we are adding New, Open, Save, Save As and Exit.



##### Figure

Now click the OK button and go back to your form. You can see the dropped down submenus when you click on the item File, as shown.



##### Figure

Finally, you can enter the code by clicking on any of the submenu items.

Simple vb program to dwesign a calculator

### [Here's the code for simple calulator](http://www.vb6.us/source-code/simple-calculator-written-vb6#comment-11115)

Take a few commands and arrange them in calculator style and start executing the below code with proper command names as specified.

Dim Op1, Op2 As Double  
Dim Opr As String

Private Sub Command1\_Click()  
Text1.Text = Text1.Text + Command1.Caption  
If cleardisplay Then  
Text1.Text = ""  
cleardisplay = False  
End If  
End Sub

Private Sub Command2\_Click()  
Text1.Text = Text1.Text + Command2.Caption  
If cleardisplay Then  
Text1.Text = ""  
cleardisplay = False  
End If

End Sub

Private Sub Command3\_Click()  
Text1.Text = Text1.Text + Command3.Caption  
If cleardisplay Then  
Text1.Text = ""  
cleardisplay = False  
End If

End Sub

Private Sub Command4\_Click()  
Text1.Text = Text1.Text + Command4.Caption  
If cleardisplay Then  
Text1.Text = ""  
cleardisplay = False  
End If

End Sub

Private Sub Command5\_Click()  
Text1.Text = Text1.Text + Command5.Caption  
If cleardisplay Then  
Text1.Text = ""  
cleardisplay = False  
End If

End Sub

Private Sub Command6\_Click()  
Text1.Text = Text1.Text + Command6.Caption  
If cleardisplay Then  
Text1.Text = ""  
cleardisplay = False  
End If

End Sub

Private Sub Command7\_Click()  
Text1.Text = Text1.Text + Command7.Caption  
If cleardisplay Then  
Text1.Text = ""  
cleardisplay = False  
End If

End Sub

Private Sub Command8\_Click()  
Text1.Text = Text1.Text + Command8.Caption  
If cleardisplay Then  
Text1.Text = ""  
cleardisplay = False  
End If

End Sub

Private Sub Command9\_Click()  
Text1.Text = Text1.Text + Command9.Caption  
If cleardisplay Then  
Text1.Text = ""  
cleardisplay = False  
End If  
End Sub

Private Sub Command10\_Click()  
Text1.Text = Text1.Text + Command10.Caption  
If cleardisplay Then  
Text1.Text = ""  
cleardisplay = False  
End If  
End Sub  
Private Sub CommandDot\_Click()  
If InStr(Text1.Text, ".") Then  
Exit Sub  
Else  
Text1.Text = Text1.Text + "."  
End If  
End Sub

Private Sub Commandcls\_Click()  
Text1.Text = ""  
End Sub

Private Sub CommandPlus\_Click()  
Op1 = Val(Text1.Text)  
Opr = "+"  
Text1.Text = ""  
End Sub

Private Sub CommandMinus\_Click()  
Op1 = Val(Text1.Text)  
Opr = "-"  
Text1.Text = ""  
End Sub

Private Sub CommandMul\_Click()  
Op1 = Val(Text1.Text)  
Opr = "\*"  
Text1.Text = ""  
End Sub

Private Sub CommandDiv\_Click()  
Op1 = Val(Text1.Text)  
Opr = "/"  
Text1.Text = ""  
End Sub

Private Sub CommandEqu\_Click()  
Op2 = Val(Text1.Text)  
Select Case Opr  
Case "+": Text1.Text = Op1 + Op2  
Case "\*": Text1.Text = Op1 \* Op2  
Case "-": Text1.Text = Op1 - Op2  
Case "/": Text1.Text = Op1 / Op2  
End Select  
End Sub

Private Sub Commandexit\_Click()  
End  
End Sub

Private Sub Form\_Load()  
Text1.Text = ""  
End Sub