Unit I: Introduction to VB

1.0 Program Design and Implementation

1.1 Design considerations

2.0 Introduction to Visual Basic

2.0 Evolution of Visual Basic

3.0 Hardware and Software Requirements of Visual Basic

4.0 Terms Often used in Visual Basic

5.0 An Application/Program in Visual Basic

6.0 Summary

1.0 Program Design and Implementation

Software design is a process of problem-solving and planning for a software solution. After the purpose and specifications of software are determined, software developers will design or employ designers to develop a plan for a solution. It includes low-level component and algorithm implementation issues as well as the architectural view. A computer program is an organized list of instructions that, when executed, causes the computer to behave in a predetermined manner. Without programs, computers are useless. Therefore, programming means designing or creating a set of instructions to ask the computer to carry out certain jobs which normally are very much faster than human beings can do.

The software requirements analysis (SRA) step of a software development process yields specifications that are used in software engineering. If the software is "semiautomated" or user centered, software design may involve user experience design yielding a story board to help determine those specifications. If the software is completely automated (meaning no user or user interface), a software design may be as simple as a flow chart or text describing a planned sequence of events. There are also semi-standard methods like Unified Modeling Language and Fundamental modeling concepts. In either case some documentation of the plan is usually the product of the design.

A software design may be platform-independent or platform-specific, depending on the availability of the technology called for by the design.

1.1 Design considerations

There are many aspects to consider in the design of a piece of software. The importance of each should reflect the goals the software is trying to achieve. Some of these aspects are:

- Compatibility - The software is able to operate with other products that are designed for interoperability with another product. For example, a piece of software may be backward-compatible with an older version of itself.
- Extensibility - New capabilities can be added to the software without major changes to the underlying architecture.
- Fault-tolerance - The software is resistant to and able to recover from component failure.
• Maintainability - The software can be restored to a specified condition within a specified period of time. For example, antivirus software may include the ability to periodically receive virus definition updates in order to maintain the software's effectiveness.
• Modularity - the resulting software comprises well defined, independent components. That leads to better maintainability. The components could be then implemented and tested in isolation before being integrated to form a desired software system. This allows division of work in a software development project.
• Packaging - Printed material such as the box and manuals should match the style designated for the target market and should enhance usability. All compatibility information should be visible on the outside of the package. All components required for use should be included in the package or specified as a requirement on the outside of the package.
• Reliability - The software is able to perform a required function under stated conditions for a specified period of time.
• Reusability - the modular components designed should capture the essence of the functionality expected out of them and no more or less. This single-minded purpose renders the components reusable wherever there are similar needs in other designs.
• Robustness - The software is able to operate under stress or tolerate unpredictable or invalid input. For example, it can be designed with a resilience to low memory conditions.
• Security - The software is able to withstand hostile acts and influences.
• Usability - The software user interface must be intuitive (and often aesthetically pleasing) to its target user/audience. In many cases, online help should be included and also carefully designed.

A software designer or architect may identify a design problem which has been solved by others before. A template or pattern describing a solution to a common problem is known as a design pattern. The reuse of such patterns can speed up the software development process, having been tested and proved in the past. Software design documentation may be reviewed or presented to allow constraints, specifications and even requirements to be adjusted prior to programming. Redesign may occur after review of a programmed simulation or prototype. It is possible to design software in the process of programming, without a plan or requirement analysis, but for more complex projects this would not be considered a professional approach. A separate design prior to programming allows for multidisciplinary designers and Subject Matter Experts (SMEs) to collaborate with highly-skilled programmers for software that is both useful and technically sound.

2.0 Introduction to Visual Basic
A Computer cannot understand any persons spoken language. A spoken language such as English, French, is simply too general and ambiguous for computers to understand. Therefore, we must adapt to the machine and learn the computer language so that the computer can understand. This is where visual basic comes into it - when we type visual basic source code into the computer, the computer processes these statements into Visual Basic language. The statements in visual basic, however, do not have multiple meanings within the same context. Like the BASIC programming language, Visual Basic was designed to be easy to learn and use. The language not only allows programmers to create simple GUI applications, but can also develop complex applications.
As long as we are familiar with Windows 95, we can easily create applications with Visual Basic. We don't have to be a windows expert, but we should feel comfortable working with menus, the mouse and the windows interface. If we have opened, closed
and re-sized windows we should have the ability to make our own Visual Basic applications.

Visual Basic is a highly popular language in the commercial world because it allows for the rapid development of Windows based programs. VB is particularly strong at creating front ends for databases. This can be done in amazing time through the use of wizards.

A more limited version of Visual Basic is also included in several other Microsoft Applications such as MS Access. With VB 6, we can create any program depending on your objective. For example, you can create educational programs to teach science, mathematics, language, history, geography and so on. You can also create financial and accounting programs to make you a more efficient accountant or financial controller. For those who like games, can create those programs as well. Programming in VB is a combination of visually arranging components or controls on a form, specifying attributes and actions of those components, and writing additional lines of code for more functionality. Since default attributes and actions are defined for the components, a simple program can be created without the programmer having to write many lines of code. Performance problems were experienced by earlier versions, but with faster computers and native code compilation this has become less of an issue.

Although programs can be compiled into native code executables from version 5 onwards, they still require the presence of runtime libraries of approximately 1 MB in size. This runtime is included by default in Windows 2000 and later, but for earlier versions of Windows like 95/98/NT it must be distributed together with the executable.

Forms are created using drag-and-drop techniques. A tool is used to place controls (e.g., text boxes, buttons, etc.) on the form (window). Controls have attributes and event handlers associated with them. Default values are provided when the control is created, but may be changed by the programmer. Many attribute values can be modified during run time based on user actions or changes in the environment, providing a dynamic application. For example, code can be inserted into the form resize event handler to reposition a control so that it remains centered on the form, expands to fill up the form, etc. By inserting code into the event handler for a keypress in a text box, the program can automatically translate the case of the text being entered, or even prevent certain characters from being inserted.

Visual Basic can create executables (EXE files), ActiveX controls, DLL files, but is primarily used to develop Windows applications and to interface web database systems. Dialog boxes with less functionality can be used to provide pop-up capabilities. Controls provide the basic functionality of the application, while programmers can insert additional logic within the appropriate event handlers. For example, a drop-down combination box will automatically display its list and allow the user to select any element. An event handler is called when an item is selected, which can then execute additional code created by the programmer to perform some action based on which element was selected, such as populating a related list.

Alternatively, a Visual Basic component can have no user interface, and instead provide ActiveX objects to other programs via Component Object Model (COM). This allows for server-side processing or an add-in module.

The language is garbage collected using reference counting, has a large library of utility objects, and has basic object oriented support. Since the more common components are included in the default project template, the programmer seldom needs to specify additional libraries. Unlike many other programming languages, Visual Basic is generally not case sensitive, although it will transform keywords into a standard case configuration and force the case of variable names to conform to the case of the entry within the symbol table entry. String comparisons are case sensitive by default, but can be made case insensitive if so desired.

The Visual Basic compiler is shared with other Visual Studio languages (C, C++), but restrictions in the IDE (Integrated Development Environment) do not allow the creation of some targets (Windows model DLL's) and threading models.
Visual Basic has the following traits which differ from C-derived languages:

- Boolean constant True has numeric value −1. This is because the Boolean data type is stored as a 16-bit signed integer. In this construct −1 evaluates to 16 binary 1s (the Boolean value True), and 0 as 16 0s (the Boolean value False). This is apparent when performing a Not operation on a 16 bit signed integer value 0 which will return the integer value −1, in other words True = Not False. This inherent functionality becomes especially useful when performing logical operations on the individual bits of an integer such as And, Or, Xor and Not. This definition of True is also consistent with BASIC since the early 1970s Microsoft BASIC implementation and is also related to the characteristics of CPU instructions at the time.

- Logical and bitwise operators are unified. This is unlike some C-derived languages (such as Perl), which have separate logical and bitwise operators. This again is a traditional feature of BASIC.

- Variable array base. Arrays are declared by specifying the upper and lower bounds in a way similar to Pascal and Fortran. It is also possible to use the Option Base statement to set the default lower bound. Use of the Option Base statement can lead to confusion when reading Visual Basic code and is best avoided by always explicitly specifying the lower bound of the array. This lower bound is not limited to 0 or 1, because it can also be set by declaration. In this way, both the lower and upper bounds are programmable. In more subscript-limited languages, the lower bound of the array is not variable. This uncommon trait does exist in Visual Basic .NET but not in VBScript.

- Relatively strong integration with the Windows operating system and the Component Object Model.

- Banker's rounding as the default behavior when converting real numbers to integers with the Round function.

- Integers are automatically promoted to reals in expressions involving the normal division operator (/) so that division of an odd integer by an even integer produces the intuitively correct result. There is a specific integer divide operator (\) which does truncate.

- By default, if a variable has not been declared or if no type declaration character is specified, the variable is of type Variant. However this can be changed with DefType statements such as DefInt, DefBool, DefVar, DefObj, DefStr. There are 12 DefType statements in total offered by Visual Basic 6.0. The default type may be overridden for a specific declaration by using a special suffix character on the variable name (# for Double, ! for Single, & for Long, % for Integer, $ for String, and @ for Currency) or using the key phrase As (type). VB can also be set in a mode that only explicitly declared variables can be used with the command Option Explicit.

2.1 Evolution of Visual Basic

VB 1.0 was introduced in 1991. The drag and drop design for creating the user interface is derived from a prototype form generator developed by Alan Cooper and his company called Tripod. Microsoft contracted with Cooper and his associates to develop Tripod into a programmable form system for Windows 3.0, under the code name Ruby (no relation to the Ruby programming language). Tripod did not include a programming language at all. Microsoft decided to combine Ruby with the Basic language to create Visual Basic.

The Ruby interface generator provided the "visual" part of Visual Basic and this was combined with the "EB" Embedded BASIC engine designed for Microsoft's abandoned "Omega" database system. Ruby also provided the ability to load dynamic link libraries containing additional controls (then called "gizmos"), which later became the VBX interface.
Timeline of Visual Basic (VB1 to VB6)
- Project ‘Thunder’ was initiated
- Visual Basic 1.0 (May 1991) was released for Windows at the Comdex/Windows World trade show in Atlanta, Georgia.

Visual Basic for MS-DOS
- Visual Basic 1.0 for DOS was released in September 1992. The language itself was not quite compatible with Visual Basic for Windows, as it was actually the next version of Microsoft's DOS-based BASIC compilers, QuickBASIC and BASIC Professional Development System. The interface used the "COW" (Character Oriented Windows) interface, using extended ASCII characters to simulate the appearance of a GUI.
- Visual Basic 2.0 was released in November 1992. The programming environment was easier to use, and its speed was improved. Notably, forms became instantiable objects, thus laying the foundational concepts of class modules as were later offered in VB4.
- Visual Basic 3.0 was released in the summer of 1993 and came in Standard and Professional versions. VB3 included version 1.1 of the Microsoft Jet Database Engine that could read and write Jet (or Access) 1.x databases.
- Visual Basic 4.0 (August 1995) was the first version that could create 32-bit as well as 16-bit Windows programs. It also introduced the ability to write non-GUI classes in Visual Basic. Incompatibilities between different releases of VB4 caused installation and operation problems. While previous versions of Visual Basic had used VBX controls, Visual Basic now used OLE controls (with files names ending in .OCX) instead. These were later to be named ActiveX controls.
- With version 5.0 (February 1997), Microsoft released Visual Basic exclusively for 32-bit versions of Windows. Programmers who preferred to write 16-bit programs were able to import programs written in Visual Basic 4.0 to Visual Basic 5.0, and Visual Basic 5.0 programs can easily be converted with Visual Basic 4.0. Visual Basic 5.0 also introduced the ability to create custom user controls, as well as the ability to compile to native Windows executable code, speeding up calculation-intensive code execution. A free, downloadable Control Creation Edition was also released for creation of ActiveX controls. It was also used as an introductory form of Visual Basic: a regular .exe project could be created and run in the IDE, but not compiled.
- Visual Basic 6.0 (Mid 1998) improved in a number of areas including the ability to create web-based applications. VB6 has entered Microsoft's "non-supported phase" as of March 2008. Although the Visual Basic 6.0 development environment is no longer supported, the runtime is supported on Windows Vista, Windows Server 2008 and Windows 7.
- Mainstream Support for Microsoft Visual Basic 6.0 ended on March 31, 2005. Extended support ended in March 2008. In response, the Visual Basic user community expressed its grave concern and lobbied users to sign a petition to keep the product alive. Microsoft has so far refused to change their position on the matter. Ironically, around this time (2005), it was exposed that Microsoft's new anti-spyware offering, Microsoft AntiSpyware (part of the GIANT Company Software purchase), was coded in Visual Basic 6.0. Its replacement, Windows Defender, was rewritten as C++ code.

3.0 Hardware and Software Requirements of Visual Basic
Hardware requirements are identical to those for CATIA V5 or ENOVIA VPLM, depending on the applications being developed, with the following exceptions:

Required components and features
- **Disk drive:** An internal or external disk drive of at least 4 GB is required to store program executables, program data, usage environment, and paging space.
- **Memory:** At least 512MB of real memory is recommended for all applications.

**Windows x86-64 64-bit platform hardware requirements**
- **Disk drive:** 15 GB
- **Memory:** 2 GB of real memory is recommended for all applications.
- **Processor:** Intel Xeon EM64T, Intel Pentium4 EM64T, and AMD Opteron 64-bit

**Software requirements:**

**Common software requirements**

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Supported Operating System Level(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Windows 32-bit</td>
<td>Windows XP Professional SP2</td>
</tr>
<tr>
<td></td>
<td>Windows Vista x86</td>
</tr>
<tr>
<td>Microsoft Windows 64-bit</td>
<td>Windows XP Professional x64 Edition SP2</td>
</tr>
<tr>
<td></td>
<td>Windows Vista x64</td>
</tr>
<tr>
<td>IBM AIX</td>
<td>AIX 5.3 Technical Level 05 Service Pack 3</td>
</tr>
<tr>
<td>Hewlett-Packard HP-UX</td>
<td>HP 11iv1 (11.11) December 2004</td>
</tr>
<tr>
<td>Sun Solaris</td>
<td>Solaris 10 HW 03/05</td>
</tr>
</tbody>
</table>

Table 1

**System requirements**

Run-time software requirements for CAA RADE V5 APIs are the same as those described in the applicable CATIA, DELMIA, and ENOVIA VPLM announcements. In addition, refer to the following:

<table>
<thead>
<tr>
<th>Applications built with CAA RADE on</th>
<th>Format Generation</th>
<th>Run on Windows XP 32-bit</th>
<th>Run on Windows XP Prof. 32-bit</th>
<th>Run on Windows Vista 32-bit</th>
<th>Run on Windows Vista x64 Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows XP 32-bit</td>
<td>32-bit</td>
<td>Yes</td>
<td>Yes*</td>
<td>Yes</td>
<td>Yes*</td>
</tr>
<tr>
<td>Windows XP Professional x64 Edition</td>
<td>64-bit 32-bit</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes*</td>
</tr>
<tr>
<td>Windows Vista 32-bit</td>
<td>32-bit</td>
<td>Yes</td>
<td>Yes*</td>
<td>Yes</td>
<td>Yes*</td>
</tr>
<tr>
<td>Windows Vista 64-bit</td>
<td>64-bit 32-bit</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes*</td>
</tr>
</tbody>
</table>

* When running on Windows 64-bit platforms, Windows 32-bit applications take advantage of Windows-32-on-Windows-64 emulation mode (WoW64).

Table 2
Software installation:

On Windows, the process of installation and deinstallation makes use of Windows-compliant tools such as Install Shield, simplifying the task for those familiar with Windows procedures and concepts. These procedures are also ported to the UNIX environment to preserve a common V5 installation interface for all supported operating systems.

4.0 Terms Often used in Visual Basic

32-bit
The number of bits that can be processed or transmitted in parallel, or the number of bits used for single element in a data format. Although this term is used throughout computing and data processing (as are 8-bit, 16-bit, and similar formulations), in VB terms, this means the number of bits used to represent memory addresses. The break between 16-bit and 32-bit processing happened with the introduction of VB5 and OCX technology.

Access Level
In VB code, the ability of other code to access it (that is, read it or write to it). The access level is determined both by how you declare the code and by the access level of the container of the code. If code can't access a containing element, then it can't access any of its contained elements either, no matter how they're declared.

Access Protocol
The software and API that allows applications and databases to communicate information. Examples include ODBC - Open DataBase Connectivity, an early protocol that is often used in conjunction with others and ADO - ActiveX Data Objects, Microsoft's protocol for accessing all kinds of information, including databases.

ActiveX
is Microsoft's specification for reusable software components. ActiveX is based on COM, the Component Object Model. The basic idea is to define exactly how software components interact and interoperate so developers can create components that work together using the definition. ActiveX components were originally called OLE Servers and ActiveX Servers and this renaming (actually for marketing rather than technical reasons) has created a lot of confusion about what they are. A lot of languages and applications support ActiveX in some way or another and Visual Basic supports it very strongly since it's one of the cornerstones of the Win32 environment.

API
is a TLA (Three Letter Acronym) for Application Program Interface. An API consists of the routines, protocols and tools that programmers must use to ensure that their programs are compatible with the software that the API is defined for. A well defined API helps applications work together by providing the same basic tools for all programmers to use. A wide variety of software from operating systems to individual components are said to have an API.

Automation Controller
Automation is a standard way to make a software object available through a defined set of interfaces. This is a great idea because the object is available to any language that follows the standard methods. The standard used in Microsoft (and therefore VB) architecture is called OLE automation. An automation controller is an application that can use the objects belonging to another application. An automation server (sometimes called an automation component) is an application that provides the programmable objects to the other applications.

Cache
A cache is a temporary information store used in both hardware (a processor chip typically includes a hardware memory cache) and software. In web programming, a cache stores the most recent web pages visited. When the 'Back' button (or other methods) are used to revisit a web page, the browser will check the cache to see if the
page is stored there and will retrieve it from the cache to save time and processing. Programmers should remember that program clients might not always retrieve a page directly from the server. This sometimes results in very subtle program bugs.

Class
The formal definition for an object and the template from which an instance of an object is created. The main purpose of the class is to define the properties and methods for the class. Although included in previous versions of Visual Basic, the class has become a key technology in VB.NET and its object-oriented programming.

Among the important ideas about classes are:

- A class can have subclasses that can inherit all or some of the characteristics of the class.
- Subclasses can also define their own methods and variables that are not part of their parent class.
- The structure of a class and its subclasses is called the class hierarchy.

Classes involve a lot of terminology. An original class, from which interface and behavior is derived, can be identified by any of these equivalent names:

- Parent class
- Superclass
- Base class

And new classes can have these names:

- Child class
- Subclass

CGI
is Common Gateway Interface. This is an early standard used to transfer information between a web server and a client over a network. For example, a form in a "shopping cart" application might contain information about a request to purchase a particular item. The information could be passed to a web server using CGI. CGI is still used a great deal, ASP is a complete alternative that works better with Visual Basic.

Client/Server
A computing model that divides processing between two (or more) processes. A client makes requests that are carried out by the server. It's important to understand that the processes could be running on the same computer but they normally run over a network. For example, when developing ASP applications, programmers often use PWS, a server that runs on the same computer with a browser client such as IE. When the same application goes into production, it normally runs over the Internet. In advanced business applications, multiple layers of clients and servers are used. This model now dominates computing and replaced the model of mainframes and 'dumb terminals' which were really only display monitors attached directly to a large mainframe computer.

In object oriented programming, a class that provides a method to another class is called the server. The class that uses the method is called the client.

Collection
The concept of a collection in Visual Basic is simply a way to group similar objects. Both Visual Basic 6 and VB.NET provide a Collection class to give you the ability to define your own collections.

So, for example, this VB 6 code snippet adds two Form1 objects to a collection and then displays a MsgBox that tells you that there are two items in the collection.

Private Sub Form_Load()
Dim myCollection As New Collection
Dim FirstForm As New Form1
Dim SecondForm As New Form1
myCollection.Add FirstForm
myCollection.Add SecondForm
MsgBox (myCollection.Count)
End Sub

**COM**

is Component Object Model. Although often associated with Microsoft, COM is an open standard that specifies how components work together and interoperate. Microsoft used COM as the basis for ActiveX and OLE. The use of the COM API ensures that a software object can be launched within your application using a wide variety of programming languages including Visual Basic. Components save a programmer from having to re-write code. A component can be large or small and can perform any kind of processing, but it must be re-usable and it must conform to set standards to for interoperability.

**Control**

In Visual Basic, the tool you use to create objects on a Visual Basic form. Controls are selected from the Toolbox and then used to draw objects on the form with the mouse pointer. It's key to realize that the control is just the tool used to create GUI objects, not the object itself.

**Cookie**

A small packet of information that is originally sent from a web server to your browser and stored on our computer. When your computer consults the originating web server again, the cookie is sent back to the server, allowing it to respond to you using information from the previous interaction. Cookies are usually used to provide customized web pages using a profile of your interests that were provided the first time you access the web server. In other words, the web server will appear to "know" you and provide what you want. Some people feel that allowing cookies is a security problem and disable them using an option provided by the browser software. As a programmer, you can't depend on the ability to use cookies all the time.

**DLL**

is Dynamic Link Library, a set of functions that can be executed, or data that can be used by a Windows application. DLL is also the file type for DLL files. For example, 'crypt32.dll' is the Crypto API32 DLL used for cryptography on Microsoft operating systems. There are hundreds and possibly thousands installed on your computer. Some DLLs are used only by a specific application, while others, such as crypt32.dll, are used by a wide variety of applications. The name refers to the fact that DLL's contain a library of functions that can be accessed (linked) on demand (dynamically) by other software.

**Encapsulation**

is the Object Oriented Programming technique that allows programmers to completely determine the relationship between objects using the object interface (the way the objects are called and the parameters passed). In other words, an object can be thought of as being "in a capsule" with the interface as the only way to communicate with the object.

The main benefits of encapsulation are that you avoid bugs because you're completely certain about how an object is being used in your program and the object can be replaced with a different one if necessary as long as the new one implements the exact same interface.

**Event Procedure**

A block of code that is called when an object is manipulated in a Visual Basic program. The manipulation can be done by a user of the program through the GUI, by the program, or through some other process such as the expiration of a time interval. For example, most Form object have a Click event. The Click Event Procedure for the form Form1 would be identified by the name Form1_Click().

**Expression**

In Visual Basic, this is a combination that evaluates to a single value. For example, the integer variable Result is given the value of an expression in the following code snippet:
Dim Result as Integer
Result = CInt((10 + CInt(vbRed) = 53 * vbThursday))
In this example, Result is assigned the value -1 which is the integer value of True in Visual Basic. To help you verify this, vbRed is equal to 255 and vbThursday is equal to 5 in Visual Basic. Expressions can be a combination of operators, constants, literal values, functions, and names of fields (columns), controls, and properties.

File Extension / File Type
In Windows, DOS and some other operating systems, one or several letters at the end of a filename. Filename extensions follow a period (dot) and indicate the type of file. For example, 'this.txt' is a plain text file, 'that.htm' or 'that.html' indicates that the file is a web page. The Windows operating system stores this association information in the Windows Registry and it can be changed using the 'File Types' dialog window provided by Windows Explorer.

Frames
A format for web documents that divides the screen into areas that can be formatted and controlled independently. Often, one frame is used to select a category while another frame shows the contents of that category.

Function
In Visual Basic, a type of subroutine that can accept an argument and returns a value assigned to the function as though it was a variable. You can code your own functions or use built-in functions provided by Visual Basic. For example, in this example, both Now and MsgBox are functions. Now returns the system time.
MsgBox(Now)

Host
A Computer or a process on a computer that provides a service to another computer or process. For example, VBScript can be ‘hosted’ by the web browser program, Internet Explorer.

Inheritance
Inheritance is the ability of one object to automatically take on the methods and properties of another object. The object that supplies the methods and properties is usually called the parent object and the object that assumes them is called the child. So, for example, in VB .NET, you will often see statements like this:

```vbnet
Public Class Form1
    Inherits System.Windows.Forms.Form
```
The parent object is System.Windows.Forms.Form and it has a large set of methods and properties that have been pre-programmed by Microsoft. Form1 is the child object and it gets to take advantage of all of the parent’s programming. The key OOP (Object Oriented Programming) behavior that was added when VB .NET was introduced is Inheritance. VB 6 supported Encapsulation and Polymorphism, but not Inheritance.

Instance
is a word seen in Object Oriented Programming explanations. It refers to a copy of an object that has been created for use by a specific program. In VB 6, for example, the statement CreateObject(objectname) will create an instance of a class (a type of object). In VB 6 and VB .NET, the keyword New in a declaration creates an instance of an object. The verb instantiate means the creation of an instance. An example in VB 6 is:

```vbnet
Dim ExcelSheet As Object
Set ExcelSheet = CreateObject("Excel.Sheet")
```

ISAPI
is the Internet Server Application Program Interface. Usually, any term that ends in the characters ‘API’ is an Application Program Interface. This is the API used by Microsoft's Internet Information Server (IIS) web server. Web applications that use ISAPI run considerably faster than those that use CGI, since they share the 'process' (programming memory space) used by the IIS web server and therefore avoid the time
consuming program load and unload process that CGI requires. A similar API used by Netscape is called NSAPI.

**Keyword**

Keywords are the words or symbols that are the elementary parts of the Visual Basic programming language. As a result, you can't use them as names in your program. Some simple examples:

Dim Dim as String
or
Dim String as String

Both of these are invalid because Dim and String are both keywords and can't be used as variable names.

**Method**

A way to identify a software function that performs an action or a service for a particular object. For example, the `Hide()` method for form `Form1` removes the form from the program display but doesn't unload it from memory. It would be coded: `Form1.Hide`

**Module**

A Module is a general term for a file containing code or information that you add to your project. Usually, a module contains program code which you write. In VB 6, modules have a .bas extension and there are just three kinds of modules: form, standard, and class. In VB.NET, modules usually have a .vb extension but others are possible, such as .xsd for a dataset module, .xml for an XML module, .htm for a web page, .txt for a text file, .xslt for an XSLT file, .css for a Style Sheet, .rpt for a Crystal Report, and others.

**Namespace**

The concept of a namespace has been around for quite a while in programming but has only become a requirement for Visual Basic programmers to know about since XML and .NET became critical technologies. The traditional definition of a namespace is a name that uniquely identifies a set of objects so there is no ambiguity when objects from different sources are used together. The type of example that you usually see is something like the Dog namespace and the Furniture namespace both have Leg objects so you can refer to a Dog.Leg or a Furniture.Leg and be very clear about which one you mean.

In practical .NET programming, however, a namespace is just the name that is used to refer to Microsoft's libraries of objects. For example, both System.Data and System.XML are typical References in default VB .NET Windows Applications and the collection of objects they contain are referred to as the System.Data namespace and the System.XML namespace.

The reason "made-up" examples like "Dog" and "Furniture" are used in other definitions is that the "ambiguity" problem really only comes up when you define your own namespace, not when you're using Microsoft's object libraries. For example, try to find object names that are duplicated between System.Data and System.XML.

When you're using XML, a namespace is a collection of element type and attribute names. These element types and attribute names are uniquely identified by the name of the XML namespace of which they are a part. In XML, a namespace is given the name of a Uniform Resource Identifier (URI) - such as a Web site's address - both because the namespace could be associated with the site and because a URI is a unique name. When it's used this way, the URI is not required to be used other than as a name and there doesn't have to be a document or XML schema at that address.

**Newsgroup**

A discussion group operated through the Internet. Newsgroups (also known as Usenet) are accessed and viewed on the web. Outlook Express (distributed by Microsoft as part of IE) supports newsgroup viewing. Newsgroups tend to be popular, fun, and alternative. See Usenet.
Object
Microsoft defines it as . . .
*a software component that exposes its properties and methods*

Halvorson (*VB.NET Step by Step*, Microsoft Press) defines it as . . .
The name of a user interface element you create on a VB form with a Toolbox control

Software that has properties and/or methods. A Document, Branch or Relationship can be an individual object, for example. Most, but not all, objects are members of a collection of some kind.

**Object Library**
A file with the .olb extension that provides information to Automation controllers (like Visual Basic) about available objects. The Visual Basic Object Browser (View menu or function key F2) will let you browse all of the object libraries available to you.

**OCX**
The file extension (and generic name) for OLE Custom control (the X must have been added because it looked cool to Microsoft Marketing types). OCX modules are independent program modules that can be accessed by other programs in a Windows environment. OCX controls replaced VBX controls written in Visual Basic. OCX, both as a marketing term and a technology, was replaced by ActiveX controls. ActiveX is backward compatible with OCX controls because ActiveX containers, such as Microsoft's Internet Explorer, can execute OCX components. OCX controls can be either 16-bit or 32-bit.

**OLE**
OLE stands for Object Linking and Embedding. This is a technology that first came on the scene along with the first really successful version of Windows: Windows 3.1. (Which was released in April 1992. Yes, Virginia, they had computers that long ago.) The first trick that OLE made possible was the creation of what is called a "compound document" or a document that has content created by more than one application. For example, a Word document containing a genuine Excel spreadsheet (not a picture, but the actual thing). The data can be provided by either "linking" or "embedding" which accounts for the name. OLE has gradually been extended to servers and networks and has gained more and more capability.

**OOP - Object Oriented Programming**
A programming architecture that emphasizes the use of objects as the fundamental building blocks of programs. This is accomplished by providing a way to create the building blocks so they include both data and functions that are accessed through an interface (these are called "properties" and "methods" in VB).

The definition of OOP has been controversial in the past because some OOP purists vehemently insisted that languages like C++ and Java were object oriented and VB 6 was not because OOP was defined (by the purists) as incorporating the three pillars: Inheritance, Polymorphism, and Encapsulation. And VB 6 never implemented inheritance. Other authorities (Dan Appleman, for example), pointed out that VB 6 was very productive for building binary reusable code blocks and therefore it was OOP enough. This controversy will die down now because VB .NET is very emphatically OOP - and most definitely includes Inheritance.

**Perl**
Is an acronym that actually expands to 'Practical Extraction and Report Language' but this doesn't do much to help you understand what it is. Although it was created for text processing, Perl has become the most popular language for writing CGI programs and was the original language of the web. People who have a lot of experience with Perl love it and swear by it. New programmers, however, tend to swear at it instead because it has a reputation for not being easy to learn. VBScript and Javascript are replacing Perl for web programming today. Perl is also used a great deal by Unix and Linux administrators for automating their maintenance work.
**Process**
Refers to a program that is currently executing, or "running" on a computer.

**Polymorphism**
is a word seen in Object Oriented Programming explanations. This is the ability to have two different objects, of two different types, that both implement the same method (polymorphism literally means "many forms"). So, for example, you might write a program for a government agency called Get License. But the license could be a dog license, a driver's license or a license to run for political office ("license to steal" ??). Visual Basic determines which one is intended by differences in the parameters used to call the objects. Both VB 6 and VB .NET provide polymorphism, but they use a different architecture to do it.

**Property**
In Visual Basic, a named attribute of an object. For example, every Toolbox object has a Name property. Properties can be set by changing them in the Properties window at design time or by program statements at run time. For example, I might change the Name property of a form Form1 with the statement:

```
Form1.Name = "MyFormName"
```

VB 6 uses Property Get, Property Set and Property Let statements to manipulate properties of objects. This syntax has been completely overhauled in VB.NET. The Get and Set syntax isn't at all the same and Let isn't supported at all.
In VB.NET a member field in a class is a property.

```vbnet
Class MyClass
    Private memberfield as String
    Public Sub classmethod()
        ' whatever this class does
    End Sub
End Class
```

**Public**
In Visual Basic .NET, the keyword in the declaration statement that makes the elements accessible from code anywhere within the same project, from other projects that reference the project, and from any assembly built from the project. But see Access Level as well on this.
Here's an example:

```
Public Class aPublicClassName
Public can be used only at module, interface, or namespace level. You can't declare an element to be Public within a procedure.
```

**Register**
Registering a DLL (Dynamic Link Library) means the system knows how to find it when an application creates an object using the DLL's ProgID. When a DLL is compiled, Visual Basic automatically registers it on that machine for you. COM depends on the Windows registry and requires all COM components to store (or 'register') information about themselves in the registry before they can be used. A unique ID is used for different components to make certain they don't clash. The ID is called a GUID, or Globally Unique IDentifier and they're calculated by compilers and other development software using a special algorithm.

**Scope**
The part of a program where a variable can be recognized and used in statements. For example, if a variable is declared (DIM statement) in the Declarations section of a form, then the variable can be used in any procedure in that form (such as the Click event for a button on the form).
State
The current condition and values in a running program. This is usually most significant in an online environment (such as a web system such as an ASP program) where the values contained in program variables will be lost unless they're saved somehow. Saving critical "state information" is a common task necessary in writing online systems.

String
Any expression that evaluates to a sequence of contiguous characters. In Visual Basic, a string is the variable type (VarType) 8.

Syntax
The word "syntax" in programming is almost the same as "grammar" in human languages. In other words, it's the rules you use to create statements. The syntax in Visual Basic must let the Visual Basic compiler 'understand' your statements to create an executable program.

This statement has incorrect syntax
a==b
because there is no "==" operation in Visual Basic. (At least, there isn't one yet! Microsoft continually adds to the language.)

URL
Uniform Resource Locator - This is the unique address of any a document on the Internet. The different parts of a URL have specific meaning.

The Parts of a URL

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Domain Name</th>
<th>Path</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>http://</td>
<td>visualbasic.about.com/ library/weekly/</td>
<td>blglossa.htm</td>
<td></td>
</tr>
</tbody>
</table>

'Protocol', for example, could be FTP:// or MailTo:// among other things.

Usenet
Usenet is a world-wide distributed discussion system. It consists of a set of 'newsgroups' with names that are classified hierarchically by subject. 'Articles' or 'messages' are posted to these newsgroups by people on computers with the appropriate software. These articles are then broadcast to other interconnected computer systems via a wide variety of networks.

VBX
The file extension (and generic name) of components used by 16-bit versions of Visual Basic (VB1 through VB4). Now obsolete, VBXs do not have two of the properties (inheritance and polymorphism) many believe are required by true object-oriented systems. Starting with VB5, OCX and then ActiveX controls became current.

Virtual Machine
A term used to describe a platform, that is, the software and operating environment, for which you are writing code. This is a key concept in VB.NET because the virtual machine that the VB 6 programmer writes to is radically different than the one the VB.NET program uses. As a starting point (but there is much more), VB.NET's virtual machine requires the presence of the CLR (Common Language Runtime). To illustrate the concept of a virtual machine platform in actual use, VB.NET provides for alternates in the Build menu Configuration Manager:
Fig 1

Web Services
Software that runs over a network and provides information services based on XML standards that are accessed through a URI (Universal Resource Identifier) address and an XML defined information interface. The standard XML technologies normally used in web services include SOAP, WSDL, UDDI and XSD.

Win32

XML
The Extensible Markup Language allows designers to create their own customized 'markup tags' for information. This makes it possible to define, transmit, validate, and interpret information between applications with greater flexibility and accuracy. The XML specification was developed by the W3C (the World Wide Web consortium - an association whose members are international corporations) but XML is used for applications far beyond the web. (Many definitions you can find on the web state that it's used only for the web, but this is a common misunderstanding. XHTML is a specific set of markup tags that are based on HTML 4.01 as well as XML that is exclusively for web pages.) VB.NET and all Microsoft .NET technologies use XML extensively.

5.0 An application/Program in Visual Basic
Hello World Example for students new to coding with Microsoft's Visual Basic 6.0 programming language. Each step in using the IDE (Integrated Development Environment) to complete the Hello World project is explained in detail.

Fig 2
Purpose of the project: To display the text “Hello World!” in a text box control, to be able to clear the text from the text box using a command button, and to exit the project using a command button. Open Visual Basic 6.0. Open a Standard .exe project. Maximize the white-background project form by clicking on the middle maximize button to the right of the Project1 title bar. Clear and Exit are buttons named cmdClear and cmdExit. The white box is a text box named txtHello.

To name the project, double-click in the project window on the Project1 icon. Look into the Properties Box where it now says (Name) Project1 and double-click on (Name). This should now highlight the text, Project1. If Project1 is highlighted just type HelloWorld (no spaces). Press the Enter key. The project is now named HelloWorld.

we could similarly highlight the form in the project window, double-click the form icon, and find the name property for the Form1 in the Properties Window, then rename it, if you so desire. I did not rename the Form1 for this Hello World demonstration. To place a text box onto your form find the text box control inside the tool box to the left of your work area. Double-click on the text box to put a text box onto your form. Use the resizing handles to size it the way you wish and drag the control using the left mouse button to move it where you would like to have it place on your form. Select the text box so that selection handles (resizing handles) appear around the text box control. In the properties window scroll to the text property and where the text reads Text1, highlight Text1 using your mouse, and delete it using the delete key. Press Enter. Now
the text box control on the form should appear to be empty. With the Text1 object still highlighted, double click on the name property so the I-beam of the cursor is inside the box where you will name the text box object. Type txtHello and press the Enter key.

To place the 3 command buttons onto your form, find the command button control object in the tool box and double-click on the command button 3 times to place three buttons onto your form. Use the left mouse in the down position to drag each command button into place. Select the first command button by clicking it with the mouse one time. Selection handles (called resizing handles) appear around the command button. When an object is selected the properties for that particular object appear in the Properties window. Double-click on the name property and type a name: cmdHello. Scroll down to the caption property and double-click caption: Type &Display Hello. The ampersand is located above the number 7 on the keyboard. Press the Enter Key.

Note: When the ampersand (&) is used before a letter in a caption, the letter following the ampersand is underlined. Any underlined letter in a caption may be used in combination with the ALT key + the underlined letter as a keyboard short-cut to access that control. A user may press ALT + D instead of clicking with the mouse on the first command button, the one with the caption Display Hello.

Now name the second command button:
Select the second command button so that you see the resizing handles. In the Properties window, double-click on (Name) and name this button cmdClear. Double-click on the Caption property in the properties window and give it a Caption of Clear. (The C should be underlined in the caption, for this project.) Press the Enter key.

(NOTE: Sometimes you will want to underline a different letter in other projects. Place the ampersand before the letter you wish to underline. Do not have the same letter underlined more than once in a project window. Two different captions should not each have an underlined D. Pick a different letter to underline when this happens. Select a letter that has not yet been used to underline in a caption).
Name the third command button:
Select the third command button so that you see the resizing handles around the third command button. In the Properties window, double-click on the (Name) property and name it cmdE&xit. Make the Caption property E&xit with the ampersand before the x so that the x will be underlined. Press the Enter key. Now you have named and labeled the Exit key.

You have opened a standard .exe project and named the project HelloWorld. The form was not renamed by me but is still called Form1. Rename the form if you wish. You have placed one text box and 3 command buttons onto your form and you have named the controls and given the command buttons separate captions. (A caption on a button is what the user sees on that button.) Text boxes do not have a caption. Use the .text property for text boxes. You have deleted default text from the text box control to make it appear empty, and have named this control, txtHello. Now you will code the command button controls.

To access the code window with a sub procedure already started for you double-click on the cmdHello button with the caption that reads Display Hello. This double-click action takes you to the code window where you see a line reading Private Sub cmdHello_Click() and another line reading End Sub. Place your cursor between these two lines and indent once using the tab key. Type txtHello.Text="Hello World!". Use the quotes this time when typing that line. Press the Enter key. On the next line indent once using the tab key and type With txtHello, then press the Enter key. On the next
line indent twice using the tab key and type .font="Arial" using the period before font and quotes around Arial. Press the Enter key. After the .font="Arial" line, on the line below indent twice using the tab key and type .FontSize=16 (no quotes but must have the period before .FontSize. Press the Enter key. On the next line indent twice using the tab key and type .ForeColor=vbBlue (no quotes, just the period before ForeColor. (You are almost through with this control). You will only indent once on the next line, using the tab key. Type End With and press the Enter key. Allow the last line of text for this sub procedure to read End Sub (no period). You have now coded the entire cmdHello sub procedure which will place text into the txtHello control when the program runs, will change the text to a 16 point Arial font and the text color (ForeColor) will be blue.

From the Visual Basic menu bar, click on View and select Object to get back to the form. You will code the sub procedure now for the second command button, the cmdClear button with the caption of Clear:

Double-click on the cmdClear button on the form. In the code window and following the line of text that reads Private Sub cmdClear_Click(), indent once using the tab key. Type txtHello.Text="", that is two quotes without a space or any content between them. This represents an empty string which appears as an empty text box when the cmdClear button is clicked or accessed from the keyboard (Alt + C). You are already through coding the cmdClear event. Let the last line of this event to be the End Sub. Go back to View, Object to return to the form.

You will code the cmdExit procedure. Double-click on the cmdExit button. In the event procedure beginning Private Sub cmdExit_Click, drop down one line, indent, and type End (no period). That's it! Return to the form (View, Object) and run your procedure (F5 button). The total code for your project looks like this:

```vbnet
Private Sub cmdClear_Click()
    txtHello.Text = ""
End Sub

Private Sub cmdExit_Click()
    End
End Sub

Private Sub cmdHello_Click()
    txtHello.Text = "Hello World!"
    With txtHello
        .Font = "Arial"
        .FontSize = 16
        .ForeColor = vbBlue
    End With
End Sub
```

Visual Basic is not a command line program so the order of the coded events does not matter in your code. No event will be triggered unless the user accesses a command button either by clicking on it or using your coded short-cut key with the Alt key. The only thing that makes a click event work is by the user "clicking" on something (or using the coded short-cut). If the user clicks on a command button and no code is written, nothing will happen either. A user event is a mouse click, a key press, dragging the mouse over a certain area, and so forth. When a user event occurs and
that object has a sub procedure coded for that user event, then the code will run. The programmer only programs what will happen “IF” a user event occurs.

6.0 Summary
In this unit we discussed about the fundamentals of Program Design and Implementation. We discussed some design considerations. Then we discussed about Visual Basic, why we use this language and evolution of VB6. We saw what software and Hardwares are required to install Visual basic. Then we discussed about the terms often used in Visual Basic. After all these things we discussed VB program and how we code the program by using VB environment.
Unit 2: Integrated Development Environment

Unit Contains

1.0 Integrated Development Environment
2.0 The project explorer windows
3.0 Form Object
4.0 Main component of the Visual Basic environment
   toolboxes, toolbar's, project window, properties window and source code window.
5.0 Adding/Removing Custom Controls to the Toolbox
6.0 Summary

1.0 The Integrated Development Environment
Learning the ins and outs of the Development Environment before you learn visual basic is somewhat like learning for a test you must know where all the functions belong and what their purpose is.

Visual Basic and its follow-up, Visual Basic for Applications (VBA - the scripting language for Microsoft Office Suite and about 100 other major Windows applications) pioneered and refined visual programming. Visual programming means that you paint the look and feel of your program and then tie together the forms, controls and overall processing with code and procedural code. The task of making the visual parts function as an effective whole program is expedited by templates, wizards and design tools within the Visual Basic IDE-Interactive Development Environ.

This is your visual development "canvas".

Figure 1 shows the Visual Basic canvas. This is where you will be working both in VBA and Visual Basic a good percentage of the time. So it will be worthwhile getting to know these Basic elements very well. As we shall see when we cover Java next time, this style/usage of multiple windows (called MDI-Multiple Document Interface by Microsoft) is a very popular IDE.. Most programming development tools have copied this layout to some degree, so learning it well will pay dividends.

Everytime you load a VB or VBA project, you will be greeted by roughly the layout shown in Figure 1 and these five GUI tools. The toolbox contains all the GUI elements/controls needed to create any VB form and the front end to all VB programs. For example, after the pointer tool there is the image control, label, textbox, frame and command button as the first five of 20 standard controls which are used constantly in VB programs. Another advantage of these basic controls is that they fill 60-90% of all your programming needs and are automatically included in the VB runtime. It is possible to add many other Microsoft supplied or third party ActiveX components/controls to this toolbox (including your own home-built control); but these added components can add significantly to the final size of your runtime.

The below diagram shows the development environment with all the important points labelled. Many of Visual basic functions work similar to Microsoft word eg the Tool Bar and the tool box is similar to other products on the market which work off a single click then drag the width of the object required. The Tool Box contains the control you placed on the form window. All of the controls that appear on the Tool Box controls on the below picture never runs out of controls as soon as you place one on the form another awaits you on the Tool Box ready to be placed as needed. First we will start with labelling the development environment.
The project explorer window

The project explorer window gives you a tree-structured view of all the files inserted into the application. You can expand these and collapse branches of the views to get more or less detail (Project explorer). The project explorer window displays forms, modules or other separators which are supported by the visual basic like class'es and Advanced Modules. If you want to select a form on its own simply double click on the project explorer window for a more detailed look. And it will display it where the Default form is located.

Properties Window
Some programmers prefer the Categorismo view of the properties window. By defaulting, the properties window displays its properties alphabetically (with the exception of the name value) when you click on the categorized button the window changes to left picture.

**The Default Layout**
When we start Visual Basic, we are provided with a VB project. A VB project is a collection of the following modules and files.

- The **global module** (that contains declaration and procedures)
- The **form module** (that contains the graphic elements of the VB application along with the instruction)
- The **general module** (that generally contains general-purpose instructions not pertaining to anything graphic on-screen)
- The **class module** (that contains the defining characteristics of a class, including its properties and methods)
- The **resource files** (that allows you to collect all of the texts and bitmaps for an application in one place)

On start up, Visual Basic will displays the following windows:

- The **Blank Form** window
- The **Project** window
- The **Properties** window

It also includes a **Toolbox** that consists of all the controls essential for developing a VB Application. Controls are tools such as boxes, buttons, labels and other objects draw on a form to get input or display output. They also add visual appeal.

**Object Methods**

- we know that each object has properties and events associated with it. A third concept associated with objects is the **method**. A method is a procedure or function that imparts some action to an object.
- As we move through the toolbox, when appropriate, we'll discuss object methods. Methods are always enacted at run-time in code. The format for invoking a method is:

  ObjectName.Method {optional arguments}

  Note: this is another use of the dot notation.

**3.0 The Form Object**

The **Form** is where the user interface is drawn. It is central to the development of Visual Basic applications

- **Form Properties:**

  **Appearance** Selects 3-D or flat appearance.
  **BackColor** Sets the form background color.
  **BorderStyle** Sets the form border to be fixed or sizeable.
  **Caption** Sets the form window title.
  **Enabled** If True, allows the form to respond to mouse and keyboard events; if False, disables form.
  **Font** Sets font type, style, size.
  **ForeColor** Sets color of text or graphics.
  **Picture** Places a bitmap picture in the form.
  **Visible** If False, hides the form.

- **Form Events:**
  **Activate** Form_Activate event is triggered when form becomes the active window.
Click Form_Click event is triggered when user clicks on form.

DoubleClick Form_DblClick event is triggered when user doubleclicks on form.

Load Form_Load event occurs when form is loaded. This is a good place to initialize variables and set any runtime properties.

• **Form Methods:**
  - **Cls** Clears all graphics and text from form. Does not clear any objects.
  - **Print** Prints text string on the form.

**Examples**

```vbnet
frmExample.Cls ' clears the form
frmExample.Print "This will print on the form"
```

### 4.0 Main component of the Visual Basic environment

**Tool box**

You may have noticed that when you click on different controls the **Properties Window** changes slightly this is due to different controls having different functions. Therefore more options are needed for example if you had a picture then you want to show an image. But if you wanted to open a internet connection you would have to fill in the remote host and other such settings. When you use the command ( broaden ) you will find that a new set of properties come up the following will provide a description and a property.

---

**Figure 3**

![Command Buttons](image)

**Command Buttons**

- We've seen the **command button** before. It is probably the most widely used control. It is used to begin, interrupt, or end a particular process.

- **Command Button Properties:**
  - **Name** The name of the object so you can call it at runtime.
  - **Appearance** Selects 3-D or flat appearance.
  - **Cancel** Allows selection of button with Esc key (only one button on a form can have this property True).
  - **Caption** String to be displayed on button.
Default  Allows selection of button with **Enter** key (only one button on a form can have this property True).

**Font**  Sets font type, style, size.

- **Command Button Events:**
  **Click**  Event triggered when button is selected either by clicking on it or by pressing the access key.

**Label Boxes**

- A **label box** is a control you use to display text that a user can't edit directly. We've seen, though, in previous examples, that the text of a label box can be changed at run-time in response to events.

- **Label Properties:**
  **Alignment**  Aligns caption within border.
  **Appearance**  Selects 3-D or flat appearance.
  **AutoSize**  If True, the label is resized to fit the text specified by the caption property. If False, the label will remain the size defined at design time and the text may be clipped.
  **BorderStyle**  Determines type of border.
  **Caption**  String to be displayed in box.
  **Font**  Sets font type, style, size.
  **WordWrap**  Works in conjunction with AutoSize property.

  If AutoSize = True, WordWrap = True, then the text will wrap and label will expand vertically to fit the Caption.
  If AutoSize = True, WordWrap = False, then the text will not wrap and the label expands horizontally to fit the Caption.
  If AutoSize = False, the text will not wrap regardless of WordWrap value.

- **Label Events:**
  **Click**  Event triggered when user clicks on a label.
  **DblClick**  Event triggered when user double-clicks on a label.

**Text Boxes**

- A **text box** is used to display information entered at design time, by a user at runtime, or assigned within code. The displayed text may be edited.

- **TextBox Properties:**
  **Appearance**  Selects 3-D or flat appearance.
  **BorderStyle**  Determines type of border.
  **Font**  Sets font type, style, size.
  **MaxLength**  Limits the length of displayed text (0 value indicates unlimited length).
  **MultiLine**  Specifies whether text box displays single line or multiple lines.
  **PasswordChar**  Hides text with a single character.
  **ScrollBars**  Specifies type of displayed scroll bar(s).
  **SelLength**  Length of selected text (run-time only).
  **SelStart**  Starting position of selected text (run-time only).
  **SelText**  Selected text (run-time only).
  **Tag**  Stores a string expression.
  **Text**  Displayed text.

- **TextBox Events:**
Change Triggered every time the Text property changes.
LostFocus Triggered when the user leaves the text box. This is a good place to examine the contents of a text box after editing.
KeyPress Triggered whenever a key is pressed. Used for key trapping, as seen in last class.

- **Text Box Methods:**

SetFocus Places the cursor in a specified text box.

Example
txtExample.SetFocus ' moves cursor to box named txtExample

Example

Password Validation
1. Start a new project. The idea of this project is to ask the user to input a password. If correct, a message box appears to validate the user. If incorrect, other options are provided.
2. Place a two command buttons, a label box, and a text box on your form so it looks something like this:

![Figure 4](image)

3. Set the properties of the form and each object.

Form1:
BorderStyle 1-Fixed Single
Caption Password Validation
Name frmPassword

Label1:
Alignment 2-Center
BorderStyle 1-Fixed Single
Caption Please Enter Your Password:
FontSize 10
FontStyle Bold

Text1:
FontSize 14
FontStyle Regular
Name txtPassword
PasswordChar *
Tag [Whatever you choose as a password]
Text [Blank]

Command1:
Caption &Validate
Default True
Name cmdValid
**Command2:**
Cancel True
Caption Exit
Name cmdExit

Your form should now look like this:

![Password Validation Form](image)

**Figure 5**

4. Attach the following code to the **cmdValid_Click** event.

```vbscript
Private Sub cmdValid_Click()
    'This procedure checks the input password
    Dim Response As Integer
    If txtPassword.Text = txtPassword.Tag Then
        'If correct, display message box
        MsgBox "You've passed security!", vbOKOnly + vbExclamation, "Access Granted"
    Else
        'If incorrect, give option to try again
        Response = MsgBox("Incorrect password", vbRetryCancel + vbCritical, "Access Denied")
        If Response = vbRetry Then
            txtPassword.SelStart = 0
            txtPassword.SelLength = Len(txtPassword.Text)
        Else
            End
        End If
    End If
    txtPassword.SetFocus
End Sub
```

This code checks the input password to see if it matches the stored value. If so, it prints an acceptance message. If incorrect, it displays a message box to that effect and asks the user if they want to try again. If Yes (Retry), another try is granted. If No (Cancel), the program is ended. Notice the use of **SelLength** and **SelStart** to highlight an incorrect entry. This allows the user to type right over the incorrect response.

5. Attach the following code to the **Form_Activate** event.

```vbscript
Private Sub Form_Activate()
    txtPassword.SetFocus
End Sub
```

6. Attach the following code to the **cmdExit_Click** event.

```vbscript
Private Sub cmdExit_Click()
    End
```
End Sub
7. Try running the program. Try both options: input correct password (note it is case
sensitive) and input incorrect password. Save your project.
If you have time, define a constant, TRYMAX = 3, and modify the code to allow
the user to have just TRYMAX attempts to get the correct password. After the
final try, inform the user you are logging him/her off. You'll also need a variable
that counts the number of tries (make it a Static variable).

Check Boxes
- **Check boxes** provide a way to make choices from a list of potential
candidates. Some, all, or none of the choices in a group may be selected.

  **Check Box Properties:**
  - **Caption** Identifying text next to box.
  - **Font** Sets font type, style, size.
  - **Value** Indicates if unchecked (0, vbUnchecked), checked (1, vbChecked), or grayed
    out (2, vbGrayed).

  **Check Box Events:**
  - **Click** Triggered when a box is clicked. Value property is automatically changed by
    Visual Basic.

Option Buttons
- **Option buttons** provide the capability to make a mutually exclusive choice
among a group of potential candidate choices. Hence, option buttons work as a
group, only one of which can have a True (or selected) value.

  **Option Button Properties:**
  - **Caption** Identifying text next to button.
  - **Font** Sets font type, style, size.
  - **Value** Indicates if selected (True) or not (False). Only one option button in a group can
    be True. One button in each group of option buttons should always be initialized to
    True at design time.

  **Option Button Events:**
  - **Click** Triggered when a button is clicked. Value property is automatically changed by
    Visual Basic.

Arrays
- Up to now, we've only worked with regular variables, each having its own unique
name. Visual Basic has powerful facilities for handling multi-dimensional
variables, or **arrays**. For now, we'll only use single, fixed-dimension arrays.

  Arrays are declared in a manner identical to that used for regular variables. For
example, to declare an integer array named 'Items', with dimension 9, at the
procedure level, we use:

  Dim Items(9) as Integer

If we want the array variables to retain their value upon leaving a procedure, we use
the keyword **Static**:

Static Items(9) as Integer

At the **form** or **module** level, in the general declarations area of the Code window,
use:
Dim Items(9) as Integer

And, at the module level, for a **global** declaration, use:

Global Items(9) as Integer

- The index on an array variable begins at 0 and ends at the dimensioned value. For example, the `items` array in the above examples has ten elements, ranging from `Items(0)` to `Items(9)`. You use array variables just like any other variable – just remember to include its name and its index. For example, to set `Item(5)` equal to 7, you simply write:

```
Item(5) = 7
```

**Control Arrays**

- With some controls, it is very useful to define **control arrays** - it depends on the application. For example, option buttons are almost always grouped in control arrays.

- Control arrays are a convenient way to handle groups of controls that perform a similar function. All of the events available to the single control are still available to the array of controls, the only difference being an argument indicating the index of the selected array element is passed to the event. Hence, instead of writing individual procedures for each control (i.e. not using control arrays), you only have to write one procedure for each array.

- Another advantage to control arrays is that you can add or delete array elements at run-time. You cannot do that with controls (objects) not in arrays. Refer to the **Load** and **Unload** statements in on-line help for the proper way to add and delete control array elements at run-time.

Two ways to create a control array:

1. Create an individual control and set desired properties. Copy the control using the editor, then paste it on the form. Visual Basic will pop-up a dialog box that will ask you if you wish to create a control array. Respond yes and the array is created.
2. Create all the controls you wish to have in the array. Assign the desired control array name to the first control. Then, try to name the second control with the same name. Visual Basic will prompt you, asking if you want to create a control array. Answer yes. Once the array is created, rename all remaining controls with that name.

- Once a control array has been created and named, elements of the array are referred to by their name and index. For example, to set the **Caption** property of element 6 of a label box array named `lblExample`, we would use:

```
lblExample(6).Caption = "This is an example"
```

We'll use control arrays in the next example.

**Frames**

- We've seen that both option buttons and check boxes work as a group. **Frames** provide a way of grouping related controls on a form. And, in the case of option buttons, frames affect how such buttons operate.

- To group controls in a frame, you first draw the frame. Then, the associated controls must be drawn in the frame. This allows you to move the frame and controls
together. And, once a control is drawn within a frame, it can be copied and pasted to create a control array within that frame. To do this, first click on the object you want to copy. **Copy** the object. Then, click on the frame. **Paste** the object. You will be asked if you want to create a control array. Answer **Yes**.

- Drawing the controls outside the frame and dragging them in, copying them into a frame, or drawing the frame around existing controls will not result in a proper grouping. It is perfectly acceptable to draw frames within other frames.
- As mentioned, frames affect how option buttons work. Option buttons within a frame work as a group, independently of option buttons in other frames. Option buttons on the form, and not in frames, work as another independent group. That is, the form is itself a frame by default. We'll see this in the next example.
- It is important to note that an independent group of option buttons is defined by physical location within frames, not according to naming convention. That is, a control array of option buttons does not work as an independent group just because it is a control array. It would only work as a group if it were the only group of option buttons within a frame or on the form. So, remember physical location, and physical location only, dictates independent operation of option button groups.

- **Frame Properties:**
  - **Caption** Title information at top of frame.
  - **Font** Sets font type, style, size.

**Example**

**Pizza Order**

1. Start a new project. We'll build a form where a pizza order can be entered by simply clicking on check boxes and option buttons.
2. Draw three frames. In the first, draw three option buttons, in the second, draw two option buttons, and in the third, draw six check boxes. Draw two option buttons on the form. Add two command buttons. Make things look something like this.

![Image of a form with three frames, containing check boxes and option buttons, and two command buttons](image.png)

3. Set the properties of the form and each control.

**Form1:**
- BorderStyle 1-Fixed Single
- Caption Pizza Order
- Name frmPizza

**Frame1:**
- Caption Option1

**Frame2:**
- Caption Option4

**Frame3:**
- Caption Crust Type

*Figure 6*
Frame3
Caption Toppings

**Exploring the Visual Basic Toolbox 3-17**

**Option1:**
Caption Small
Name optSize
Value True

**Option2:**
Caption Medium
Name optSize (yes, create a control array)

**Option3:**
Caption Large
Name optSize

**Option4:**
Caption Thin Crust
Name optCrust
Value True

**Option5:**
Caption Thick Crust
Name optCrust (yes, create a control array)

**Option6:**
Caption Eat In
Name optWhere
Value True

**Option7:**
Caption Take Out
Name optWhere (yes, create a control array)

**Check1:**
Caption Extra Cheese
Name chkTop

**Check2:**
Caption Mushrooms
Name chkTop (yes, create a control array)

**Check3:**
Caption Black Olives
Name chkTop

**Check4:**
Caption Onions
Name chkTop

**Check5:**
Caption Green Peppers
Name chkTop

**Check6:**
Caption Tomatoes
Name chkTop

**Command1:**
Caption &Build Pizza
Name cmdBuild

**Command2:**
Caption E&xit
Name cmdExit

The form should look like this now:
4. Declare the following variables in the **general declarations** area:

   Option Explicit
   Dim PizzaSize As String
   Dim PizzaCrust As String
   Dim PizzaWhere As String

   This makes the size, crust, and location variables global to the form.

5. Attach this code to the **Form_Load** procedure. This initializes the pizza size, crust, and eating location.

   ```vbscript
   Private Sub Form_Load()
   'Initialize pizza parameters
   PizzaSize = "Small"
   PizzaCrust = "Thin Crust"
   PizzaWhere = "Eat In"
   End Sub
   ```

   Here, the global variables are initialized to their default values, corresponding to the default option buttons.

6. Attach this code to the three option button array **Click** events. Note the use of the Index variable:

   ```vbscript
   Private Sub optSize_Click(Index As Integer)
   'Read pizza size
   PizzaSize = optSize(Index).Caption
   End Sub
   ```

   ```vbscript
   Private Sub optCrust_Click(Index As Integer)
   'Read crust type
   PizzaCrust = optCrust(Index).Caption
   End Sub
   ```

   ```vbscript
   Private Sub optWhere_Click(Index As Integer)
   'Read pizza eating location
   PizzaWhere = optWhere(Index).Caption
   End Sub
   ```

   In each of these routines, when an option button is clicked, the value of the corresponding button’s caption is loaded into the respective variable.

7. Attach this code to the **cmdBuild_Click** event.
Private Sub cmdBuild_Click()
    'This procedure builds a message box that displays your
    'pizza type
    Dim Message As String
    Dim I As Integer
    Message = PizzaWhere + vbCrLf
    Message = Message + PizzaSize + " Pizza" + vbCrLf
    Message = Message + PizzaCrust + vbCrLf
    For I = 0 To 5
        If chkTop(I).Value = vbChecked Then Message = Message + chkTop(I).Caption + vbCrLf
    Next I
    MsgBox Message, vbOKOnly, "Your Pizza"
End Sub

This code forms the first part of a message for a message box by concatenating the pizza size, crust type, and eating location (vbCr is a symbolic constant representing a 'carriage return' that puts each piece of ordering information on a separate line). Next, the code cycles through the six topping check boxes and adds any checked information to the message. The code then displays the pizza order in a message box.

8. Attach this code to the cmdExit_Click event.

Private Sub cmdExit_Click()
    End
End

9. Get the application working. Notice how the different selection buttons work in their individual groups. Save your project.

10. If you have time, try these modifications:
    A. Add a new program button that resets the order form to the initial default values. You'll have to reinitialize the three global variables, reset all check boxes to unchecked, and reset all three option button groups to their default values.
    B. Modify the code so that if no toppings are selected, the message "Cheese Only" appears on the order form. You'll need to figure out a way to see if no check boxes were checked.

List Boxes

- A list box displays a list of items from which the user can select one or more items. If the number of items exceeds the number that can be displayed, a scroll bar is automatically added.

- List Box Properties:
  Appearance Selects 3-D or flat appearance.
  List Array of items in list box.
  ListCount Number of items in list.
  ListIndex The number of the most recently selected item in list. If no item is selected, ListIndex = -1.
  MultiSelect Controls how items may be selected (0-no multiple selection allowed, 1-multiple selection allowed, 2-group selection allowed).
  Selected Array with elements set equal to True or False, depending on whether corresponding list item is selected.
  Sorted True means items are sorted in 'Ascii' order, else items appear in order added.
  Text Text of most recently selected item.

- List Box Events:
  Click Event triggered when item in list is clicked.
  DblClick Event triggered when item in list is double-clicked. Primary way used to process selection.
• **List Box Methods:**
  
  **AddItem** Allows you to insert item in list.
  
  **Clear** Removes all items from list box.
  
  **RemoveItem** Removes item from list box, as identified by index of item to remove.

**Examples**

```vbnet
lstExample.AddItem "This is an added item" ' adds text string to list
lstExample.Clear ' clears the list box
lstExample.RemoveItem 4 ' removes lstExample.List(4) from list box
```

• Items in a list box are usually initialized in a Form_Load procedure. It's always a good idea to **Clear** a list box before initializing it.

• You've seen list boxes before. In the standard 'Open File' window, the Directory box is a list box with MultiSelect equal to zero.

**Combo Boxes**

• The **combo box** is similar to the list box. The differences are a combo box includes a text box on top of a list box and only allows selection of one item. In some cases, the user can type in an alternate response.

• **Combo Box Properties:**
  
  Combo box properties are nearly identical to those of the list box, with the deletion of the MultiSelect property and the addition of a Style property.
  
  **Appearance** Selects 3-D or flat appearance.
  
  **List** Array of items in list box portion.
  
  **ListCount** Number of items in list.
  
  **ListIndex** The number of the most recently selected item in list. If no item is selected, ListIndex = -1.
  
  **Sorted** True means items are sorted in 'Ascii' order, else items appear in order added.
  
  **Style** Selects the combo box form. Style = 0, Dropdown combo; user can change selection.
  
  Style = 1, Simple combo; user can change selection.
  
  Style = 2, Dropdown combo; user cannot change selection.
  
  **Text** Text of most recently selected item.

• **Combo Box Events:**
  
  **Click** Event triggered when item in list is clicked.
  
  **DblClick** Event triggered when item in list is double-clicked.
  
  Primary way used to process selection.

• **Combo Box Methods:**
  
  **AddItem** Allows you to insert item in list.
  
  **Clear** Removes all items from list box.
  
  **RemoveItem** Removes item from list box, as identified by index of item to remove.

**Examples**

```vbnet
cboExample.AddItem "This is an added item" ' adds text string to list
cboExample.Clear ' clears the combo box
cboExample.RemoveItem 4 ' removes cboExample.List(4) from list box
```

• You've seen combo boxes before. In the standard 'Open File' window, the File Name box is a combo box of Style 2, while the Drive box is a combo box of Style 3.

**Example**

**Flight Planner**

1. Start a new project. In this example, you select a destination city, a seat location, and a meal preference for airline passengers.
2. Place a list box, two combo boxes, three label boxes and two command buttons on the form. The form should appear similar to this:

![Form1](image)

**Figure 8**

3. Set the form and object properties:

**Form1:**
- BorderStyle 1-Fixed Single
- Caption Flight Planner
- Name frmFlight

**List1:**
- Name lstCities
- Sorted True

**Combo1:**
- Name cboSeat
- Style 2-Dropdown List

  (After setting properties for this combo box, resize it until it is large enough to hold 4 to 5 entries.)

**Label1:**
- Caption Destination City

**Label2:**
- Caption Seat Location

**Label3:**
- Caption Meal Preference

**Command1:**
- Caption &Assign
- Name cmdAssign

**Command2:**
- Caption E&xit
- Name cmdExit

Now, the form should look like this:
4. Attach this code to the Form_Load procedure:

```vbscript
Private Sub Form_Load()
' Add city names to list box
lstCities.Clear
lstCities.AddItem "San Diego"
lstCities.AddItem "Los Angeles"
lstCities.AddItem "Orange County"
lstCities.AddItem "Ontario"
lstCities.AddItem "Bakersfield"
lstCities.AddItem "Oakland"
lstCities.AddItem "Sacramento"
lstCities.AddItem "San Jose"
lstCities.AddItem "San Francisco"
lstCities.AddItem "Eureka"
lstCities.AddItem "Eugene"
lstCities.AddItem "Portland"
lstCities.AddItem "Spokane"
lstCities.AddItem "Seattle"
lstCities.ListIndex = 0
' Add seat types to first combo box
cboSeat.AddItem "Aisle"
cboSeat.AddItem "Middle"
cboSeat.AddItem "Window"
cboSeat.ListIndex = 0
' Add meal types to second combo box
cboMeal.AddItem "Chicken"
cboMeal.AddItem "Mystery Meat"
cboMeal.AddItem "Kosher"
cboMeal.AddItem "Vegetarian"
cboMeal.AddItem "Fruit Plate"
cboMeal.Text = "No Preference"
End Sub
```

This code simply initializes the list box and the list box portions of the two combo boxes.

5. Attach this code to the cmdAssign_Click event:

```vbscript
Private Sub cmdAssign_Click()
' Build message box that gives your assignment
```

Dim Message As String
Message = "Destination: " + lstCities.Text + vbCrLf
Message = Message + "Seat Location: " + cboSeat.Text + vbCrLf
Message = Message + "Meal: " + cboMeal.Text + vbCrLf
MsgBox Message, vbOKOnly + vbInformation, "Your Assignment"
End Sub

When the Assign button is clicked, this code forms a message box message by concatenating the selected city (from the list box lstCities), seat choice (from cboSeat), and the meal preference (from cboMeal).

6. Attach this code to the cmdExit_Click event:
Private Sub cmdExit_Click()
End
End Sub

7. Run the application. Save the project.

Line Tool
- The line tool creates simple straight line segments of various width and color.
  Together with the shape tool discussed next, you can use this tool to 'dress up' your application.
- **Line Tool Properties:**
  - **BorderColor** Determines the line color.
  - **BorderStyle** Determines the line 'shape'. Lines can be transparent, solid, dashed, dotted, and combinations.
  - **BorderWidth** Determines line width.
  - There are no events or methods associated with the line tool.
  - Since the line tool lies in the middle-layer of the form display, any lines drawn will be obscured by all controls except the shape tool or image box.

Shape Tool
- The shape tool can create circles, ovals, squares, rectangles, and rounded squares and rectangles. Colors can be used and various fill patterns are available.
- **Shape Tool Properties:**
  - **BackColor** Determines the background color of the shape (only used when FillStyle not Solid).
  - **BackStyle** Determines whether the background is transparent or opaque.
  - **BorderColor** Determines the color of the shape's outline.
  - **BorderStyle** Determines the style of the shape's outline. The border can be transparent, solid, dashed, dotted, and combinations.
  - **BorderWidth** Determines the width of the shape border line.
  - **FillColor** Defines the interior color of the shape.
  - **FillStyle** Determines the interior pattern of a shape. Some choices are: solid, transparent, cross, etc.
  - **Shape** Determines whether the shape is a square, rectangle, circle, or some other choice.
  - Like the line tool, events and methods are not used with the shape tool.
• Shapes are covered by all objects except perhaps line tools and image boxes (depends on their Z-order) and printed or drawn information. This is a good feature in that you usually use shapes to contain a group of control objects and you'd want them to lie on top of the shape.

**Horizontal and Vertical Scroll Bars**

- Horizontal and vertical **scroll bars** are widely used in Windows applications. Scroll bars provide an intuitive way to move through a list of information and make great input devices.
- Both type of scroll bars are comprised of three areas that can be clicked, or dragged, to change the scroll bar value. Those areas are:

![figure 9](image)

Clicking an **end arrow** increments the **scroll box** a small amount, clicking the **bar area** increments the scroll box a large amount, and dragging the scroll box (thumb) provides continuous motion. Using the properties of scroll bars, we can completely specify how one works. The scroll box position is the only output information from a scroll bar.

**Scroll Bar Properties:**

- **Large Change** Increment added to or subtracted from the scroll bar **Value** property when the bar area is clicked.
- **Max** The value of the horizontal scroll bar at the far right and the value of the vertical scroll bar at the bottom. Can range from -32,768 to 32,767.
- **Min** The other extreme value - the horizontal scroll bar at the left and the vertical scroll bar at the top. Can range from -32,768 to 32,767.
- **Small Change** The increment added to or subtracted from the scroll bar **Value** property when either of the scroll arrows is clicked.
- **Value** The current position of the scroll box (thumb) within the scroll bar. If you set this in code, Visual Basic moves the scroll box to the proper position.

Properties for horizontal scroll bar:

![figure 10](image)

Properties for vertical scroll bar:
A couple of important notes about scroll bars:

1. Note that although the extreme values are called Min and Max, they do not necessarily represent minimum and maximum values. There is nothing to keep the Min value from being greater than the Max value. In fact, with vertical scroll bars, this is the usual case. Visual Basic automatically adjusts the sign on the SmallChange and LargeChange properties to insure proper movement of the scroll box from one extreme to the other.

2. If you ever change the Value, Min, or Max properties in code, make sure Value is at all times between Min and Max or and the program will stop with an error message.

Scroll Bar Events:
- Change Event is triggered after the scroll box’s position has been modified. Use this event to retrieve the Value property after any changes in the scroll bar.
- Scroll Event triggered continuously whenever the scroll box is being moved.

Picture Boxes
- The picture box allows you to place graphics information on a form. It is best suited for dynamic environments - for example, when doing animation.
- Picture boxes lie in the top layer of the form display. They behave very much like small forms within a form, possessing most of the same properties as a form.

Picture Box Properties:
- AutoSize If True, box adjusts its size to fit the displayed graphic.
- Font Sets the font size, style, and size of any printing done in the picture box.
- Picture Establishes the graphics file to display in the picture box.

Picture Box Events:
- Click Triggered when a picture box is clicked.
- DblClick Triggered when a picture box is double-clicked.

Picture Box Methods:
- Cls Clears the picture box.
- Print Prints information to the picture box.
Examples
picExample.Cls ' clears the box picExample
picExample.Print "a picture box" ' prints text string to picture box

- **Picture Box LoadPicture Procedure:**
  An important function when using picture boxes is the `LoadPicture` procedure. It is used to set the `Picture` property of a picture box at run-time.

Example
```
picExample.Picture = LoadPicture("c:\pix\sample.bmp")
```
This command loads the graphics file `c:\pix\sample.bmp` into the `picture` property of the `picExample` picture box. The argument in the `LoadPicture` function must be a legal, complete path and file name, else your program will stop with an error message.

- Five types of graphics files can be loaded into a picture box:
  - **Bitmap** An image represented by pixels and stored as a collection of bits in which each bit corresponds to one pixel. Usually has a `.bmp` extension. Appears in original size.
  - **Icon** A special type of bitmap file of maximum 32 x 32 size. Has a `.ico` extension.
  - **Metafile** A file that stores an image as a collection of graphical objects (lines, circles, polygons) rather than pixels. Metafiles preserve an image more accurately than bitmaps when resized. Has a `.wmf` extension. Resizes itself to fit the picture box area.
  - **JPEG** JPEG (Joint Photographic Experts Group) is a compressed bitmap format which supports 8 and 24 bit color. It is popular on the Internet. Has a `.jpg` extension and scales nicely.
  - **GIF** GIF (Graphic Interchange Format) is a compressed bitmap format originally developed by CompuServe. It supports up to 256 colors and is popular on the Internet. Has a `.gif` extension and scales nicely.

- **Image Boxes**
  - An image box is very similar to a picture box in that it allows you to place graphics information on a form. Image boxes are more suited for static situations - that is, cases where no modifications will be done to the displayed graphics.
  - Image boxes appear in the middle-layer of form display, hence they could be obscured by picture boxes and other objects. Image box graphics can be resized by using the `Stretch` property.

- **Image Box Properties:**
  - **Picture** Establishes the graphics file to display in the image box.
  - **Stretch** If False, the image box resizes itself to fit the graphic. If True, the graphic resizes to fit the control area.

- **Image Box Events:**
  - **Click** Triggered when a image box is clicked.
  - **DbClick** Triggered when a image box is double-clicked.

- The image box does not support any methods, however it does use the `LoadPicture` function. It is used in exactly the same manner as the picture box uses it. And image boxes can load the same file types: bitmap (.bmp), icon (.ico), metafiles (.wmf), GIF files (.gif), and JPEG files (.jpg). With Stretch = True, all three graphic types will expand to fit the image box area.
Example: Picture and Image Boxes
1. Start a new project. Draw one picture box and one image box.
2. Set the Picture property of the picture and image box to the same file. If you have graphics files installed with Visual Basic, bitmap files can be found in the bitmaps folder, icon files in the icons folder, and metafiles are in the metafile folder.
3. Notice what happens as you resize the two boxes. Notice the layer effect when you move one box on top of the other. Notice the effect of the image box Stretch property.

Drive List Box
- The drive list box control allows a user to select a valid disk drive at run-time. It displays the available drives in a drop-down combo box. No code is needed to load a drive list box; Visual Basic does this for us. We use the box to get the current drive identification.
- Drive Contains the name of the currently selected drive.
- Change Triggered whenever the user or program changes the drive selection.

Directory List Box
- The directory list box displays an ordered, hierarchical list of the user's disk directories and subdirectories. The directory structure is displayed in a list box. Like, the drive list box, little coding is needed to use the directory list box – Visual Basic does most of the work for us.

File List Box
- The file list box locates and lists files in the directory specified by its Path property at run-time. You may select the types of files you want to display in the file list box.
- FileName Contains the currently selected file name.
- Path Contains the current path directory.
- Pattern Contains a string that determines which files will be displayed. It supports the use of * and ? wildcard characters. For example, using *.dat only displays files with the .dat extension.
- File List Box Events:
  - DblClick Triggered whenever a file name is double-clicked.
  - PathChange Triggered whenever the path changes in a file list box.
- You can also use the MultiSelect property of the file list box to allow multiple file selection.

Synchronizing the Drive, Directory, and File List Boxes
- The drive, directory, and file list boxes are almost always used together to obtain a file name. As such, it is important that their operation be synchronized to insure the displayed information is always consistent.

- When the drive selection is changed (drive box Change event), you should update the directory path. For example, if the drive box is named drvExample and the directory box is dirExample, use the code:
When the directory selection is changed (directory box Change event), you should update the displayed file names. With a file box named filExample, this code is:

```
filExample.Path = dirExample.Path
```

Once all of the selections have been made and you want the file name, you need to form a text string that correctly and completely specifies the file identifier. This string concatenates the drive, directory, and file name information. This should be an easy task, except for one problem. The problem involves the backslash (\) character. If you are at the root directory of your drive, the path name ends with a backslash. If you are not at the root directory, there is no backslash at the end of the path name and you have to add one before tacking on the file name.

Example code for concatenating the available information into a proper file name and then loading it into an image box is:

```
Dim YourFile as String
If Right(filExample.Path,1) = "\" Then
    YourFile = filExample.Path + filExample.FileName
Else
    YourFile = filExample.Path + "\" + filExample.FileName
End If
imgExample.Picture = LoadPicture(YourFile)
```

Note we only use properties of the file list box. The drive and directory box properties are only used to create changes in the file list box via code.

**Common Dialog Boxes**

- custom file access routines. Two common file access routines in Windows-based applications are the Open File and Save File operations. Fortunately, you don't have to build these routines.

- To give the user a standard interface for common operations in Windows-based applications, Visual Basic provides a set of common dialog boxes, two of which are the Open and Save As dialog boxes. Such boxes are familiar to any Windows user and give your application a professional look. And, with Windows 95, some context-sensitive help is available while the box is displayed. Appendix II lists many symbolic constants used with common dialog boxes.

- The Common Dialog control is a ‘custom control’ which means we have to make sure some other files are present to use it. In normal setup configurations, Visual Basic does this automatically. If the common dialog box does not appear in the Visual Basic toolbox, you need to add it. This is done by selecting Components under the Project menu. When the selection box appears, click on Microsoft Common Dialog Control, then click OK.

- The common dialog tool, although it appears on your form, is invisible at run-time. You cannot control where the common dialog box appears on your screen. The tool is invoked at run-time using one of five ‘Show’ methods. These methods are: Method Common Dialog Box
ShowOpen Open dialog box
ShowSave Save As dialog box
ShowColor Color dialog box
ShowFont Font dialog box
ShowPrinter Printer dialog box

- The format for establishing a common dialog box named cdlExample so that an Open box appears is:

cdlExample.ShowOpen

closed in some manner. Common dialog boxes are system modal.

**toolbars** which manage and control all of VB/VBA. Yes indeed there are lots of commands and operations to learn - Visual basic has sprouted like a weed in the past three years. Most readers will be familiar with many of the menu and icons. File, Edit, View, Run, Window, Help menus should be familiar to any Word Perfect, Excel, or Netscape users. Likewise icons for File Open, File Save, Cut, Copy, Paste, Find, Run programs, Pause Program, Halt Program can be seen along the toolbar. Along with built in wizards and custom command centres in some controls, this is the heart of VB.

**Summary of Major Visual Basic Canvas Features**

1) The toolbox icon - toggles the main toolbox of form controls on and off.

2) The form icon in the Project Explorer. Click on this and the active Form will appear in the edit window.

3) The toolbar handle. Every toolbar has one. Pull on it to move/reposition the toolbar anywhere on the screen.

4) The Project Explorer icon - toggle this to make the Project Explorer appear or disappear.

5) Property sheet icon - toggle this to make the Property sheet appear or disappear.

**5.0 Adding Custom Controls to the Toolbox**

Creating a toolbar in VB requires the use of two controls. The first is the ImageList control, which contains the images that will be used for the toolbar buttons. The second control is the Toolbar itself. These controls are two of nine controls that are part of Microsoft Windows Common Controls. To make these controls accessible to your VB project, go to the Project menu in the VB IDE and select Components:
This brings up the Components dialog box, where you must check Microsoft Windows Common Controls 6.0 (SP6), as shown below.
Click OK. The ImageList and Toolbar, along with the TreeView, ListView, Slider, StatusBar, ProgressBar, TabStrip, and ImageCombo will appear in your toolbox as shown below. (These other controls will be covered in separate topics; for now we will focus just on the ImageList and Toolbar.)

Toolbox with the Windows Common Controls added:

---

6.0 Summary: In this unit we studied About the Development Environment, The project explorer windows, how to create a form, Understanding the tool bar, Introducing Source code, Command Button properties, Recognise and understand the function of the main component of the Visual Basic environment eg. toolbar's, toolboxes, project window, properties window and most importantly the source code window, we have explained each and every component of toolbox with their properties as well as events. we explained these with some examples.
Block 1: VB at a Glance

Unit 3- Organizing the toolbox

Unit Contains

1. Structure of a Visual Basic Application
2. Steps in Developing Application
3. Drawing the User Interface and Setting Properties
4. Create and save a new program
5. Opening an existing Visual Basic project
6. Opening a new visual basic file & Inserting Source code
7. Running and viewing the project in detail
   7.1 There are several ways to run your program:
   7.2 There are several ways to stop your program
   7.3 Revise the program
   7.4 Examine the program in Break mode
   7.5 Exit
8. Creating Visual Basic Executable Files
   8.1 The Visual Basic Package & Deployment Wizard
9. Creating a Stand-Alone Visual Basic Application
10. Summary

1. Structure of a Visual Basic Application

Project (.VBP, .MAK)

<table>
<thead>
<tr>
<th>Form 1 (.FRM)</th>
<th>Form 2 (.FRM)</th>
<th>Form 3 (.FRM)</th>
<th>Module 1 (.BAS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control 1</td>
<td>Control 1</td>
<td>Control 1</td>
<td></td>
</tr>
<tr>
<td>Control 2</td>
<td>Control 2</td>
<td>Control 2</td>
<td></td>
</tr>
<tr>
<td>Control 3</td>
<td>Control 3</td>
<td>Control 3</td>
<td></td>
</tr>
</tbody>
</table>

Application (Project) is made up of:

- **Forms** - Windows that you create for user interface
- **Controls** - Graphical features drawn on forms to allow user interaction (text boxes, labels, scroll bars, command buttons, etc.) (Forms and Controls are **objects**.)
- **Properties** - Every characteristic of a form or control is specified by a property. Example properties include names, captions, size, color, position, and contents. Visual Basic applies default properties. You can change properties at design time or run time.
- **Methods** - Built-in procedure that can be invoked to impart some action to a particular object.
- **Event Procedures** - Code related to some object. This is the code that is executed when a certain event occurs.
- **General Procedures** - Code not related to objects. This code must be invoked by the application.
• **Modules** - Collection of general procedures, variable declarations, and constant definitions used by application.

**2 Steps in Developing Application**
• There are three primary steps involved in building a Visual Basic application:

1. **Draw** the user **interface**
2. **Assign properties** to controls
3. **Attach code** to controls

We’ll look at each step.

**3 Drawing the User Interface and Setting Properties**

Visual Basic operates in three modes.
• **Design** mode - used to build application
• **Run** mode - used to run the application
• **Break** mode - application halted and debugger is available

We focus here on the **design** mode.
• Six windows appear when you start Visual Basic.
• The **Main Window** consists of the title bar, menu bar, and toolbar. The title bar indicates the project name, the current Visual Basic operating mode, and the current form. The menu bar has drop-down menus from which you control the operation of the Visual Basic environment. The toolbar has buttons that provide shortcuts to some of the menu options. The main window also shows the location of the current form relative to the upper left corner of the screen (measured in twips) and the width and length of the current form.

⇒ The **Form Window** is central to developing Visual Basic applications. It is where you draw your application.
The Toolbox is the selection menu for controls used in your application.

The Properties Window is used to establish initial property values for objects. The drop-down box at the top of the window lists all objects in the current form. Two views are available: Alphabetic and Categorized. Under this box are the available properties for the currently selected object.
The **Form Layout Window** shows where (upon program execution) your form will be displayed relative to your monitor’s screen:

![Form Layout Window](image)

The **Project Window** displays a list of all forms and modules making up your application. You can also obtain a view of the **Form** or **Code** windows (window containing the actual Basic coding) from the Project window:

![Project Window](image)
As mentioned, the user interface is ‘drawn’ in the form window. There are two ways to place controls on a form:
1. Double-click the tool in the toolbox and it is created with a default size on the form. You can then move it or resize it.
2. Click the tool in the toolbox, then move the mouse pointer to the form window. The cursor changes to a crosshair. Place the crosshair at the upper left corner of where you want the control to be, press the left mouse button and hold it down while dragging the cursor toward the lower right corner. When you release the mouse button, the control is drawn.

To move a control you have drawn, click the object in the form window and drag it to the new location. Release the mouse button.

To resize a control, click the object so that it is select and sizing handles appear. Use these handles to resize the object.

Setting Properties of Objects at Design Time
- Each form and control has properties assigned to it by default when you start up a new project. There are two ways to display the properties of an object.
  The first way is to click on the object (form or control) in the form window. Then, click on the Properties Window or the Properties Window button in the tool bar.
  The second way is to first click on the Properties Window. Then, select the object from the Object box in the Properties Window. Shown is the Properties Window for the stopwatch application:
The drop-down box at the top of the Properties Window is the **Object** box. It displays the name of each object in the application as well as its type. This display shows the **Form** object. The **Properties** list is directly below this box. In this list, you can scroll through the list of properties for the selected object. You may select a property by clicking on it. Properties can be changed by typing a new value or choosing from a list of predefined settings (available as a drop down list). Properties can be viewed in two ways: **Alphabetic** and **Categorized**. A very important property for each object is its **name**. The name is used by Visual Basic to refer to a particular object in code.

- A convention has been established for naming Visual Basic objects. This convention is to use a three letter prefix (depending on the object) followed by a name you assign. A few of the prefixes are:
  
  **Object Prefix**  
  **Example**

<table>
<thead>
<tr>
<th>Form</th>
<th>frm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>cmd, btn</td>
</tr>
<tr>
<td>Command</td>
<td>cmdExit, btnStart</td>
</tr>
<tr>
<td>Label</td>
<td>lbl</td>
</tr>
<tr>
<td>Text Box</td>
<td>txt</td>
</tr>
<tr>
<td>Menu</td>
<td>mnu</td>
</tr>
<tr>
<td>Check Box</td>
<td>chk</td>
</tr>
</tbody>
</table>

- Object names can be up to 40 characters long, must start with a letter, must contain only letters, numbers, and the underscore (_) character. Names are used in setting properties at run time and also in establishing procedure names for object events.

**Setting Properties at Run Time**

- You can also set or modify properties while your application is running. To do this, you must write some code. The code format is:

  ```vbnet
  ObjectName.Property = NewValue
  ```
Such a format is referred to as dot notation. For example, to change the BackColor property of a form name `frmStart`, we’d type:

```
frmStart.BackColor = BLUE
```

**How Names are Used in Object Events**

- The names you assign to objects are used by Visual Basic to set up a framework of event-driven procedures for you to add code to. The format for each of these subroutines (all object procedures in Visual Basic are subroutines) is:

```
Sub ObjectName_Event (Optional Arguments)
...
End Sub
```

- Visual Basic provides the **Sub** line with its arguments (if any) and the **End Sub** statement. You provide any needed code.

### 4.0 Create and save a new program

1. Start VB.
2. Find Visual Basic on the Windows Start menu and start it.
3. The large VB window appears with a New Project dialog box (if there is no dialog box, bring it up by File > New Project).
4. Select Standard Exe. The VB window shows Project1 with an empty Form1.
5. Name the project.
6. Select Project > Project1 Properties. In the Project Properties dialog box, change Project1 to your project name, say Counter.
7. The title bar on the VB window changes to match.
8. Name the form and set the form caption.
9. The Properties window at the right of the VB window shows the properties for the form. The first property at the top of the list is Name. Change the name to, say, frmCounter.
10. The title bar in the Form window changes to match.
11. Still in the Properties window, scroll down to Caption. Change the caption to the same name as the project, say Counter.
12. The title bar on the form itself changes to match.
13. Save the form (you should always name and save the form before you save the project).

Select File > Save frmCounter As .... The Save Form As dialog box appears. The **Save in:** textbox shows the name of the folder where the form will be saved. You should create a new folder for this project. Navigate to the folder where you want to keep all your VB projects and click on the new folder icon (a picture of a folder with a little highlight). After you create the new folder, select that folder. Its name should appear in the **Save in:** box. The form file name `frmCounter.frm` should appear in the **File name:** box (if not, type in the correct name). Click on **Save**. The form is saved.

14. Save the project (be sure to name and save the form before you save the project).

Select File > Save Project As .... The Save Project As dialog box appears. The **Save in:** textbox should show the correct folder (the one you just created for the form). The **File name:** box should show the correct name (`Counter.vbp` in this example). Click on **Save**. The project is saved.

After you name and save the form and the project, it is a good idea to exit and then restart VB to make sure you can restore your project --- before you invest a lot of work in it.
5.0 Opening an existing Visual Basic project.
Microsoft have included some freebies with visual basic to show its capabilities and functions. Dismantling or modifying these sample projects is a good way to understand what is happening at runtime. These files can be located at your default directory \SAMPLES/.
To Open these projects choose 'Open Project' from the 'File' menu. Then Double click on the samples folder to open the directory then Double click on any project to load it. Steps are as follows:

1. Start VB.
   Find Visual Basic on the Windows Start menu and start it.
   The large VB window appears with a New Project dialog box. This time choose the Existing tab (if there is no dialog box, bring it up by File > Open Project).
   Navigate to the folder you created earlier. It should contain a .vbp file for the project you created earlier (Counter.vbp in this example). Select that file.
   Alternatively, instead of using the Start menu, you can navigate to the project folder using My Computer or Windows Explorer. Then double-clicking on the .vbp file or the .frm file starts VB and opens that project.
   The restored project appears in the VB window. The project name appears in the title bar.

2. Open the form.
   An icon for a folder of forms appears in the Project window on the right side of the VB window. Open this folder and select the form you saved. Click on the View Code and View Object icons to display the code or form layout windows.
   Now you are ready to add controls and code to the form. At any time, you can use File > Save frmCounter and File > Save Project to save your changes to the form and the project without going through the Save As ... dialogs.

6.0 Opening a new visual basic file & Inserting Source code
From looking at the examples it time to make your own application. Choose 'New Project' from the 'File' menu. Use the blank form1 to design a simple interface for an estate agents database, have some textboxes for names and other details. Insert some controls and make it look professional. Textboxes can be used to store there name and other details, make sure you put a picture box in for a picture of the house.
Now insert the following source code for your application.

```
Private Sub Form_Load()
    Picture1.Picture = LoadPicture("C:\Program Files\VB\Graphics\Icons\Misc\MISC42.ico")
End Sub
```

7.0 Running and viewing the project in detail.
Once an application is loaded it can be run by click on the icon from the toolbar, to pause press and to terminate use .

Once a project is loaded, the name of the form(s) that it contains is displayed in the project window. To view a form in design mode, select the form required by clicking with the mouse to highlight its name, then clicking on the view form button.
In this example the project has been loaded and the maillist.frm has been selected for viewing. This Ms Mail example project uses 6 forms and 1 modules.

In Design mode, when the form is viewed, the code attached to any screen object may be inspected by double clicking on that object. The screen shots below show the interface of the Ms Mail example (\samples\Comtool\VBMail\MaILLST.FRM) to view the code for this form select from the project window item.

Private Sub SetupOptionForm(BasePic As Control)

    BasePic.Top = 0
    BasePic.Left = 0
    BasePic.Visible = True
    BasePic.enabled = True
    OKBt.Top = BasePic.Height + 120
    Me.width = BasePic.Width + 120
    Me.Heigh = OkBt.Top + OkBt.Height + 495

End Sub

7.1 There are several ways to run your program:
- Press the F5 key.
- On the VB menu bar, Run > Start
- On the VB toolbar, click the VB Run icon (the arrow)

The program window appears, looking much like the form you designed. The controls on the window are active. While the program is running, it behaves like any other window on the desktop: you can move it, minimize it, etc.

The indicator on the VB window title bar changes from design mode to run mode. Many items in the VB window disappear during run mode and many menu and toolbar operations not enabled.

7.2 There are several ways to stop your program:
- In your program’s window, click the Exit button or menu (if you provided one)
- In your program window’s title bar, click the Close button (VB always provides one)
- On the VB menu bar, Run > Stop
- On the VB toolbar, click the VB Stop icon (the box)

The indicator on the VB window title bar changes from run mode to design mode. Items in the VB window reappear, and menu and toolbar operations are enabled again
7.3 Revise the program
In design mode, you can add or remove controls, change control properties and revise code. Your changes will be in effect when you run the program again. It is not necessary to use the Save operations before your next program run. It is only necessary to save before you exit VB.

7.4 Examine the program in Break mode
When the program is running, there are several ways to put VB in break mode:
- On the VB menu bar, Run > Break
- On the VB toolbar, click the Break icon (the vertical bars)
- Your program reaches a breakpoint or experiences an error

7.5 Exit
There are several ways to exit VB.
- On the menu bar, File > Exit
- On the VB window title bar, click the Close button
VB will warn you if you have unsaved work.

Saving your visual basic project
Save your work to disk. Use the Windows Explorer or any desktop windows to check that all files have been saved. There should be one Visual Basic Project (.VBP) file and separate Form (.FRM) and Module (.BAS) files for each form and module used in the current project.

8.0 Creating Visual Basic Executable Files
- Up to now, to run any of the applications created, we needed Visual Basic. The goal of creating an application is to let others (without Visual Basic) use it. This is accomplished by creating an executable version of the application.

- Before creating an executable, you should make sure your program is free of bugs and operating as desired. Save all forms, modules, and project files. Any later changes to the application will require re-creating the executable file.

- The executable file will have the extension .exe. To create an exe file for your application, select Make [Project name] exe from Visual Basic’s File menu. This will display the Make EXE File dialog box, where you name the exe file. To open the Options box, click that button. The EXE Options dialog box will appear:
We’ll only concern ourselves with two pieces of information in this box: Title and Icon. The Title is the name you want to give your application. It does not have to be the same as the Project name. The Icon is selected from icons assigned to form(s) in your application. The selected icon is used to identify the application everywhere it is needed in Windows 95.

Once you have selected these options, return to the Make EXE File dialog box, select a directory (best to use the same directory your application files are in) and name for your executable file. Click OK and the exe file is created.

Use Windows Explorer to confirm creation of the file. And, while there, doubleclick the executable file name and the program will run!

8.1 The Visual Basic Package & Deployment Wizard

If you gave someone a copy of your exe file and they tried to run it, it wouldn’t work (unless they have Visual Basic installed also). The reason it wouldn’t run is that the executable file also needs some ancillary files (primarily, so-called dynamic link libraries) to run properly. These libraries provide most of the code associated with keeping things on a form working properly.

So to allow others to run your application, you need to give them the executable file (exe) and at least two dynamic link libraries. Unfortunately, these dynamic link libraries take up over 1 Meg of disk space, so it’s hard to move those around on a floppy disk.

Visual Basic solves this ‘distribution problem’ by providing a very powerful tool called the Visual Basic Package & Deployment Wizard. This wizard is installed along with Visual Basic.

The Package & Deployment Wizard prepares your application for distribution. It helps you determine which files to distribute, creates a Setup program (written in Visual Basic) that works like all other Windows Setup programs (setup.exe),
compresses all required files to save disk space, and writes the compressed files to the distribution media of choice, usually floppy disk(s).

- To start the Package & Deployment Wizard, click the Start button in Windows, then find the Visual Basic program folder - click on Visual Basic Tools, then choose Package & DeploymentWizard. The setup follows several steps. The directions provided with each step pertain to the simple applications we develop in class. For more complicated examples, you need to modify the directions, especially regarding what files you need to distribute with your application.

9.0 Creating a Stand-Alone Visual Basic Application

**Step 1. Initial Information.** Enter the path and file name for your project file (.vbp). Click the ellipsis (...) to browse vbp files. If an executable (.exe) file does not exist, one will be created. Click the ‘Package’ button to continue. If you have previously saved a setup package for the selected project, it will load the package file created during that session.

**Step 2. Package Type.** Choose the Standard Setup Package (we want a standard setup program). Click Next to continue.

**Step 3. Package Folder.** Select a directory where you want the application distribution package to be saved. Click Next to continue. Click Back to return to the previous step.

**Step 4. Included Files.** The Package & Deployment Wizard will list all files it believes are required for your application to function properly. If your application requires any files not found by the wizard (for example, external data files you have created), you would add them to the setup list here (click Add). To continue, click Next. Click Back to return to the previous step.

**Step 5. Cab Options.** Distribution files are called cab files (have a cab extension). You can choose a Single cab file written to your hard drive (if you use CD ROM distribution), or Multiple cab files (to allow distribution on floppy disks). If you choose, Multiple, you
also specify the capacity of the disks you will use to write your distribution file(s). Make your choice. Click Next to Continue. Click Back to return to the previous step.

**Step 6. Installation Title.** Enter a title you want your application to have. Click Next to Continue. Click Back to return to previous step.

**Step 7. Start Menu Items.** This step determines where your installed application will be located on the user's Start menu. We will use the default choice. Click Next to Continue. Click Back to return to previous step.

**Step 8. Install Locations.** The wizard gives you an opportunity to change the locations of installed files. Click Next to Continue. Click Back to return to previous step.

**Step 9. Shared Files.** Some files in your application may be shared by other applications. Shared files will not be removed if the application is uninstalled. Decide if you have shared files. Click Next to Continue. Click Back to return to previous step.

**Step 10. Finished!** Provide a name for saving the script for this wizard session (a file that saves answers to all the questions you just answered). Click Finish to Continue. Click Back to return to previous step.

**Step 11. Write Distribution Media.** This is not a step in the wizard, but one you must take. The cab files (distribution files) the wizard wrote must now be copied to your distribution media. If you wrote a single cab file (for CD ROM), copy that file, the setup.exe file (the setup application), and the setup.lst file to your CD ROM. If you wrote multiple files (for floppy disks), copy the setup.exe, setup.lst, and first cab file (1 at end of file name) to floppy number 1. Copy the second cab file (2 at end of file name) to floppy number 2. Copy all subsequent cab files to as many floppies as needed. Properly label all floppies.

- To install the application using the distribution CD ROM or floppy disk(s), a user simply puts CD ROM or floppy number 1 in a drive. Then, through the Windows Explorer, run the setup.exe program. The user will be taken through the installation procedure step-by-step. The procedure is nearly identical to the installation process for all Microsoft products.

- The Package & Deployment Wizard is a very powerful tool. We've only looked at using it for simple applications. As your programming skills begin to include database access and other advanced topics, you will need to become familiar with other files that should be distributed with your applications.

**10. Summary:**
In this unit we studied what is the structure of Visual Basic Application. How to make a program and project. How to save, run, open and exit a project. How to make an .exe file of our project and how this exe file can be run on a system who has not contain visual basic.

References:


Unit 1: Data types, variables, constants and operators

1. Variables
   1.1 Explicit Declaration
   1.2 Using Option Explicit statement
   1.3 Scope of Variables
      a) Local Variables
      b) Static Variables
      c) Module Level Variables
      d) Public vs Local Variables

2. Data types
   2.1 Visual Basic Statements and Expressions

3. Operators
   3.1 Arithmetical Operators
   3.2 Relational Operators
   3.3 Logical Operators
   3.4 Visual Basic Functions

4. Constants

5. Visual Basic Branching
   5.0 If Statements
   5.1 Select Case
   5.2 GoTo Statement

1 Variables
Variables are the memory locations which are used to store values temporarily. A defined naming strategy has to be followed while naming a variable. A variable name must begin with an alphabet letter and should not exceed 255 characters. It must be unique within the same scope. It should not contain any special character like %, &, !, #, @ or $.

There are many ways of declaring variables in Visual Basic. Depending on where the variables are declared and how they are declared, we can determine how they can be used by our application. The different ways of declaring variables in Visual Basic are listed below and elucidated.

1.1 Explicit Declaration
1.2 Using Option Explicit statement
1.3 Scope of Variables

1.1 Explicit Declaration
Declaring a variable tells Visual Basic to reserve space in memory. It is not must that a variable should be declared before using it. Automatically whenever Visual Basic encounters a new variable, it assigns the default variable type and value. This is called implicit declaration. Though this type of declaration is easier for the user, to have more control over the variables, it is advisable to declare them explicitly. The variables are declared with a Dim statement to name the variable and its type. The As type clause in the Dim statement allows to define the data type or object type of the variable. This is called explicit declaration.
Syntax

Dim variable [As Type]

For example,

Dim strName As String
Dim intCounter As Integer

1.2 Using Option Explicit statement

It may be convenient to declare variables implicitly, but it can lead to errors that may not be recognized at run time. Say, for example a variable by name intcount is used implicitly and is assigned to a value. In the next step, this field is incremented by 1 by the following statement

Intcount = intcont + 1

This calculation will result in intcount yielding a value of 1 as intcount would have been initialized to zero. This is because the intcount variable has been mityped as incont in the right hand side of the second variable. But Visual Basic does not see this as a mistake and considers it to be new variable and therefore gives a wrong result.

In Visual Basic, to prevent errors of this nature, we can declare a variable by adding the following statement to the general declaration section of the Form:

Option Explicit

This forces the user to declare all the variables. The Option Explicit statement checks in the module for usage of any undeclared variables and reports an error to the user. The user can thus rectify the error on seeing this error message.

The Option Explicit statement can be explicitly placed in the general declaration section of each module using the following steps.

- Click Options item in the Tools menu
- Click the Editor tab in the Options dialog box
- Check Require Variable Declaration option and then click the OK button

1.3 Scope of variables

A variable is scoped to a procedure-level (local) or module-level variable depending on how it is declared. The scope of a variable, procedure or object determines which part of the code in our application are aware of the variable's existence. A variable is declared in general declaration section of e Form, and hence is available to all the procedures. Local variables are recognized only in the procedure in which they are declared. They can be declared with Dim and Static keywords. If we want a variable to be available to all of the procedures within the same module, or to all the procedures in an application, a variable is declared with broader scope.

a) Local Variables

A local variable is one that is declared inside a procedure. This variable is only available to the code inside the procedure and can be declared using the Dim statements as given below.

Dim sum As Integer

The local variables exist as long as the procedure in which they are declared, is executing. Once a procedure is executed, the values of its local variables are lost and the memory used by these variables is freed and can be reclaimed. Variables that are declared with keyword Dim exist only as long as the procedure is being executed.
b) Static Variables
Static variables are not reinitialized each time Visual Invokes a procedure and therefore retains or preserves value even when a procedure ends. In case we need to keep track of the number of times a command button in an application is clicked, a static counter variable has to be declared. These static variables are also ideal for making controls alternately visible or invisible. A static variable is declared as given below.

```
Static intPermanent As Integer
```

Variables have a lifetime in addition to scope. The values in a module-level and public variables are preserved for the lifetime of an application whereas local variables declared with Dim exist only while the procedure in which they are declared is still being executed. The value of a local variable can be preserved using the Static keyword. The following procedure calculates the running total by adding new values to the previous values stored in the static variable value.

```
Function RunningTotal ( )
Static Accumulate
Accumulate = Accumulate + num
RunningTotal = Accumulate
End Function
```

If the variable Accumulate was declared with Dim instead of static, the previously accumulated values would not be preserved across calls to the procedure, and the procedure would return the same value with which it was called. To make all variables in a procedure static, the Static keyword is placed at the beginning of the procedure heading as given in the below statement.

```
Static Function RunningTotal ( )
```

Example
The following is an example of an event procedure for a CommandButton that counts and displays the number of clicks made.

```
Private Sub Command1_Click ( )
Static Counter As Integer
Counter = Counter = 1
Print Counter
End Sub
```

The first time we click the CommandButton, the Counter starts with its default value of zero. Visual Basic then adds 1 to it and prints the result.

c) Module Level Variables
A module level variable is available to all the procedures in the module. They are declared using the Public or the Private keyword.

```
Public ans As Integer
Private Temp As Integer
```

Declaring a variable with Public keyword makes it available throughout the application even for the modules. At the module-level there is no difference between Dim and Private, but Private is preferred because it makes the code easier to read. Public variable should not be declared within a procedure. Public variables in different
modules can share the same name and they can be differentiated in code. For example, if the public integer variable intY is declared in both Form1 and Module 1 of a project it can be referred as Form1.intY and Module1.intY

d) Public vs Local Variables
A variable can have the same name and different scope. For example, we can have a public variable named R and within a procedure we can declare a local variable R. References to the name R within the procedure would access the local variable and references to R outside the procedure would access the public variable.

- We're now ready to attach code to our application. As objects are added to the form, Visual Basic automatically builds a framework of all event procedures. We simply add code to the event procedures we want our application to respond to. But before we do this, we need to discuss variables.

- Variables are used by Visual Basic to hold information needed by your application. Rules used in naming variables:
  - No more than 40 characters
  - They may include letters, numbers, and underscore (_)
  - The first character must be a letter
  - You cannot use a reserved word (word needed by Visual Basic)

- There are three ways for a variable to be typed (declared):
  1. Default
  2. Implicit
  3. Explicit

- If variables are not implicitly or explicitly typed, they are assigned the variant type by default. The variant data type is a special type used by Visual Basic that can contain numeric, string, or date data.

- To implicitly type a variable, use the corresponding suffix shown above in the data type table. For example
  TextValue$ = "This is a string" creates a string variable, while Amount% = 300 creates an integer variable.

- There are many advantages to explicitly typing variables. Primarily, we insure all computations are properly done, mistyped variable names are easily spotted, and Visual Basic will take care of insuring consistency in upper and lower case letters used in variable names. Because of these advantages, and because it is good programming practice, we will explicitly type all variables.

- To explicitly type a variable, you must first determine its scope. There are four levels of scope:
  - Procedure level
  - Procedure level, static
  - Form and module level
  - Global level

- Within a procedure, variables are declared using the Dim statement:

  Dim MyInt as Integer
  Dim MyDouble as Double
  Dim MyString, YourString as String
  Procedure level variables declared in this manner do not retain their value once a procedure terminates.
To make a procedure level variable retain its value upon exiting the procedure, replace the Dim keyword with Static:
Static MyInt as Integer
Static MyDouble as Double

Form (module) level variables retain their value and are available to all procedures within that form (module). Form (module) level variables are declared in the declarations part of the general object in the form's (module's) code window. The Dim keyword is used:
Dim MyInt as Integer
Dim MyDate as Date

Global level variables retain their value and are available to all procedures within an application. Module level variables are declared in the declarations part of the general object of a module's code window. (It is advisable to keep all global variables in one module.) Use the Global keyword:
Global MyInt as Integer
Global MyDate as Date

What happens if you declare a variable with the same name in two or more places? More local variables shadow (are accessed in preference to) less local variables. For example, if a variable MyInt is defined as Global in a module and declared local in a routine MyRoutine, while in MyRoutine, the local value of MyInt is accessed. Outside MyRoutine, the global value of MyInt is accessed.

Example of Variable Scope:

```
Module1
Global X as Integer

Form1
Dim Y as Integer
Sub Routine1()
    Dim A as Double
    ...
End Sub

Sub Routine2()
    Static B as Double
    ...
End Sub

Form2
Dim Z as Single
Sub Routine3()
    Dim C as String
    ...
End Sub
```

Procedure Routine1 has access to X, Y, and A (loses value upon termination)
Procedure Routine2 has access to X, Y, and B (retains value)
Procedure Routine3 has access to X, Z, and C (loses value)
2 Visual Basic Data Types
By default Visual Basic variables are of variant data types. The variant data type can store numeric, date/time or string data. When a variable is declared, a data type is supplied for it that determines the kind of data they can store. The fundamental data types in Visual Basic including variant are integer, long, single, double, string, currency, byte and boolean. Visual Basic supports a vast array of data types. Each data type has limits to the kind of information and the minimum and maximum values it can hold. In addition, some types can interchange with some other types. A list of Visual Basic's simple data types are given below.

1. Numeric
   - **Byte**
     - Store integer values in the range of 0 - 255
   - **Integer**
     - Store integer values in the range of (-32,768) - (+ 32,767)
   - **Long**
     - Store integer values in the range of (-2,147,483,468) - (+ 2,147,483,468)
   - **Single**
     - Store floating point value in the range of (-3.4x10^-38) - (+ 3.4x10^38)
   - **Double**
     - Store large floating value which exceeding the single data type value
   - **Currency**
     - Store monetary values. It supports 4 digits to the right of decimal point and 15 digits to the left

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

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

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

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

2.1 Visual Basic Statements and Expressions
- The simplest statement is the assignment statement. It consists of a variable name, followed by the assignment operator (=), followed by some sort of expression.

**Examples**:
- `StartTime = Now`
- `Explorer.Caption = "Captain Spaulding"`
- `BitCount = ByteCount * 8`
- `Energy = Mass * LIGHTSPEED ^ 2`
- `NetWorth = Assets - Liabilities`

The assignment statement stores information.

- Statements normally take up a single line with no terminator. Statements can be stacked by using a colon (:) to separate them.
- Example:

  `StartTime = Now : EndTime = StartTime + 10`
(Be careful stacking statements, especially with If/End If structures. You may not get the response you desire.)

- If a statement is very long, it may be continued to the next line using the continuation character, an underscore (_). Example:

  ```vba
  Months = Log(Final * IntRate / Deposit + 1) _ / Log(1 + IntRate)
  ```

- Comment statements begin with the keyword Rem or a single quote ('). For example:

  ```vba
  Rem This is a remark
  ' This is also a remark
  x = 2 * y ' another way to write a remark or comment
  ```

You, as a programmer, should decide how much to comment your code. Consider such factors as reuse, your audience, and the legacy of your code.

### 3 Visual Basic Operators

#### 3.1 Arithmetical Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add</td>
<td>5+5</td>
<td>10</td>
</tr>
<tr>
<td>-</td>
<td>Substract</td>
<td>10-5</td>
<td>5</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
<td>25/5</td>
<td>5</td>
</tr>
<tr>
<td>\</td>
<td>Integer Division</td>
<td>20\3</td>
<td>6</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
<td>5*4</td>
<td>20</td>
</tr>
<tr>
<td>^</td>
<td>Exponent (power of)</td>
<td>3^3</td>
<td>27</td>
</tr>
<tr>
<td>Mod</td>
<td>Remainder of division</td>
<td>20 Mod 6</td>
<td>2</td>
</tr>
<tr>
<td>&amp;</td>
<td>String concatenation</td>
<td>&quot;George&quot;&amp;&quot;Bush&quot;</td>
<td>&quot;George Bush&quot;</td>
</tr>
</tbody>
</table>

- Parentheses around expressions can change precedence.
- To concatenate two strings, use the & symbol or the + symbol:

  ```vba
  lblTime.Caption = "The current time is" & Format(Now, "hh:mm")
  txtSample.Text = "Hook this " + "to this"
  ```

#### 3.2 Relational Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>10&gt;8</td>
<td>True</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>10&lt;8</td>
<td>False</td>
</tr>
</tbody>
</table>
### 3.3 Logical Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td>Operation will be true if either of the operands is true</td>
</tr>
<tr>
<td>AND</td>
<td>Operation will be true only if both the operands are true</td>
</tr>
</tbody>
</table>

- The **Not** operator simply negates an operand.
- Logical operators follow arithmetic operators in precedence.

### 3.4 Visual Basic Functions

Functions, which can return a value, have the following structure:

```vbnet
[Static] Function name(argument list) [As return value]  
statements  
End Function
```

For instance:

```vbnet
Function cube(number As Integer) As Integer  
cube = number * number * number  
End Function
```

Functions are called as follows:

- `value = cube(5)`
- `cube 5`

- Visual Basic offers a rich assortment of built-in **functions**. The on-line help utility will give you information on any or all of these functions and their use. Some examples are:

#### Function Value Returned

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs</td>
<td>Absolute value of a number</td>
</tr>
<tr>
<td>Asc</td>
<td>ASCII or ANSI code of a character</td>
</tr>
<tr>
<td>Chr</td>
<td>Character corresponding to a given ASCII or ANSI code</td>
</tr>
<tr>
<td>Cos</td>
<td>Cosine of an angle</td>
</tr>
<tr>
<td>Date</td>
<td>Current date as a text string</td>
</tr>
<tr>
<td>Format</td>
<td>Date or number converted to a text string</td>
</tr>
<tr>
<td>Left</td>
<td>Selected left side of a text string</td>
</tr>
<tr>
<td>Len</td>
<td>Number of characters in a text string</td>
</tr>
<tr>
<td>Mid</td>
<td>Selected portion of a text string</td>
</tr>
<tr>
<td>Now</td>
<td>Current time and date</td>
</tr>
<tr>
<td>Right</td>
<td>Selected right end of a text string</td>
</tr>
<tr>
<td>Rnd</td>
<td>Random number</td>
</tr>
<tr>
<td>Sin</td>
<td>Sine of an angle</td>
</tr>
</tbody>
</table>
A Closer Look at the Rnd Function

- In writing games and learning software, we use the Rnd function to introduce randomness. This insures different results each time you try a program. The Visual Basic function Rnd returns a single precision, random number between 0 and 1 (actually greater than or equal to 0 and less than 1). To produce random integers (I) between Imin and Imax, use the formula:

\[ I = \text{Int}((\text{Imax} - \text{Imin} + 1) \times \text{Rnd}) + \text{Imin} \]

- The random number generator in Visual Basic must be seeded. A Seed value initializes the generator. The Randomize statement is used to do this:

```
Randomize Seed
```

If you use the same Seed each time you run your application, the same sequence of random numbers will be generated. To insure you get different numbers every time you use your application (preferred for games), use the Timer function to seed the generator:

```
Randomize Timer
```

Place this statement in the Form_Load event procedure.

- **Examples:**
  
  To roll a six-sided die, the number of spots would be computed using:
  
  \[ \text{NumberSpots} = \text{Int}(6 \times \text{Rnd}) + 1 \]
  
  To randomly choose a number between 100 and 200, use:
  
  \[ \text{Number} = \text{Int}(101 \times \text{Rnd}) + 100 \]

**Example Savings Account**

1. Start a new project. The idea of this project is to determine how much you save by making monthly deposits into a savings account. For those interested, the mathematical formula used is:

\[ F = D \left\{ \frac{(1 + I)M - 1}{I} \right\} \]

where

- \( F \) - Final amount
- \( D \) - Monthly deposit amount
- \( I \) - Monthly interest rate
- \( M \) - Number of months

2. Place 4 label boxes, 4 text boxes, and 2 command buttons on the form. It should look something like this:
3. Set the properties of the form and each object.

**Form1:**
- BorderStyle 1-Fixed Single
- Caption Savings Account
- Name frmSavings

**Label1:**
- Caption Monthly Deposit

**Label2:**
- Caption Yearly Interest

**Label3:**
- Caption Number of Months

**Label4:**
- Caption Final Balance

**Text1:**
- Text [Blank]
- Name txtDeposit

**Text2:**
- Text [Blank]
- Name txtInterest

**Text3:**
- Text [Blank]
- Name txtMonths

**Text4:**
- Text [Blank]
- Name txtFinal

**Command1:**
- Caption &Calculate
- Name cmdCalculate

**Command2:**
- Caption E&xit
- Name cmdExit

Now, your form should look like this:
4. Declare four variables in the **general declarations** area of your form. This makes them available to all the form procedures:

```
Option Explicit
Dim Deposit As Single
Dim Interest As Single
Dim Months As Single
Dim Final As Single
```

The **Option Explicit** statement forces us to declare all variables.

5. Attach code to the **cmdCalculate** command button **Click** event.

```
Private Sub cmdCalculate_Click()
    Dim IntRate As Single
    'Read values from text boxes
    Deposit = Val(txtDeposit.Text)
    Interest = Val(txtInterest.Text)
    IntRate = Interest / 1200
    Months = Val(txtMonths.Text)
    'Compute final value and put in text box
    Final = Deposit * ((1 + IntRate) ^ Months - 1) / IntRate
    txtFinal.Text = Format(Final, "#####0.00")
End Sub
```

This code reads the three input values (monthly deposit, interest rate, number of months) from the text boxes, computes the final balance using the provided formula, and puts that result in a text box.

6. Attach code to the **cmdExit** command button **Click** event.

```
Private Sub cmdExit_Click()
    End
End Sub
```

7. Play with the program. Make sure it works properly. Save the project.

### 4.0 Visual Basic Symbolic Constants

- Many times in Visual Basic, functions and objects require data arguments that affect their operation and return values you want to read and interpret. These arguments and values are constant numerical data and difficult to interpret based on just the numerical
value. To make these constants more understandable, Visual Basic assigns names to the most widely used values—these are called **symbolic constants**.

- As an example, to set the background color of a form named `frmExample` to blue, we could type:
  
  ```vbscript
  frmExample.BackColor = 0xFF0000  
  or, we could use the symbolic constant for the blue color (`vbBlue`):
  frmExample.BackColor = vbBlue
  ```

- It is strongly suggested that the symbolic constants be used instead of the numeric values, when possible. You should agree that `vbBlue` means more than the value `0xFF0000` when selecting the background color in the above example. You do not need to do anything to define the symbolic constants—they are built into Visual Basic.

**Defining Your Own Constants**

Constants are declared as follows:

```
Const name [As Type] = value
eg. Const i As Integer = 10
```

- You can also define your own constants for use in Visual Basic. The format for defining a constant named `PI` with a value `3.14159` is:

```
Const PI = 3.14159
```

- **User-defined constants** should be written in all upper case letters to distinguish them from variables. The scope of constants is established the same way a variables’ scope is. That is, if defined within a procedure, they are local to the procedure. If defined in the general declarations of a form, they are global to the form. To make constants global to an application, use the format:

```
Global Const PI = 3.14159
```

**5. Visual Basic Branching**

**5.1 If Statements**

- **Branching** statements are used to cause certain actions within a program if a certain condition is met.
- The simplest is the `If/Then` statement:

```
If Balance - Check < 0 Then Print "You are overdrawn"
```

Here, if and only if `Balance - Check` is less than zero, the statement "You are overdrawn" is printed.

- You can also have `If/Then/Else/End If` blocks to allow multiple statements:

```
If Balance - Check < 0 Then
  Print "You are overdrawn"
  Print "Authorities have been notified"
End If
```

In this case, if `Balance - Check` is less than zero, two lines of information are printed.

- Or, `If/Then/Else/End If` blocks:

  The basic structure is as follows:
  
  ```vbscript
  If condition1 Then
  statements
  ```
Else If condition2 Then
statements
Else
statements
End If

The Else If portion is optional, as is the Else part.

Example

If Balance - Check < 0 Then
Print "You are overdrawn"
Print "Authorities have been notified"
Else
Balance = Balance - Check
End If

Here, the same two lines are printed if you are overdrawn (Balance - Check < 0), but, if you are not overdrawn (Else), your new Balance is computed.

• Or, we can add the ElseIf statement:

If Balance - Check < 0 Then
Print "You are overdrawn"
Print "Authorities have been notified"
ElseIf Balance - Check = 0 Then
Print "Whew! You barely made it"
Balance = 0
Else
Balance = Balance - Check
End If

Now, one more condition is added. If your Balance equals the Check amount (ElseIf Balance - Check = 0), a different message appears.

• In using branching statements, make sure you consider all viable possibilities in the If/Else/End If structure. Also, be aware that each If and ElseIf in a block is tested sequentially. The first time an If test is met, the code associated with that condition is executed and the If block is exited. If a later condition is also True, it will never be considered.

Key Trapping
• Note in the previous example, there is nothing to prevent the user from typing in meaningless characters (for example, letters) into the text boxes expecting numerical data. Whenever getting input from a user, we want to limit the available keys they can press. The process of intercepting unacceptable keystrokes is key trapping.

• Key trapping is done in the KeyPress procedure of an object. Such a procedure has the form (for a text box named txtText):

Sub txtText_KeyPress (KeyAscii as Integer)
.
.
.
End Sub

What happens in this procedure is that every time a key is pressed in the corresponding text box, the ASCII code for the pressed key is passed to this procedure in the argument list (i.e. KeyAscii). If KeyAscii is an acceptable value, we would do
nothing. However, if KeyAscii is not acceptable, we would set KeyAscii equal to zero and exit the procedure. Doing this has the same result of not pressing a key at all. ASCII values for all keys are available via the on-line help in Visual Basic. And some keys are also defined by symbolic constants. Where possible, we will use symbolic constants; else, we will use the ASCII values.

- As an example, say we have a text box (named `txtExample`) and we only want to be able to enter upper case letters (ASCII codes 65 through 90, or, correspondingly, symbolic constants `vbKeyA` through `vbKeyZ`). The key press procedure would look like (the `Beep` causes an audible tone if an incorrect key is pressed):

```vba
Sub txtExample_KeyPress(KeyAscii as Integer)
If KeyAscii >= vbKeyA And KeyAscii <= vbKeyZ Then
    Exit Sub
Else
    KeyAscii = 0
    Beep
End If
End Sub
```

- In key trapping, it's advisable to always allow the backspace key (ASCII code 8: symbolic constant `vbKeyBack`) to pass through the key press event. Else, you will not be able to edit the text box properly.

**Savings Account - Key Trapping**

1. Note the acceptable ASCII codes are 48 through 57 (numbers), 46 (the decimal point), and 8 (the backspace key). In the code, we use symbolic constants for the numbers and backspace key. Such a constant does not exist for the decimal point, so we will define one with the following line in the general declarations area:

```vba
Const vbKeyDecPt = 46
```

2. Add the following code to the three procedures:

```vba
Private Sub txtDeposit_KeyPress (KeyAscii As Integer)
    'Only allow number keys, decimal point, or backspace
    If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or
        KeyAscii = vbKeyDecPt Or KeyAscii = vbKeyBack Then
        Exit Sub
    Else
        KeyAscii = 0
        Beep
    End If
End Sub

Private Sub txtInterest_KeyPress (KeyAscii As Integer)
    'Only allow number keys, decimal point, or backspace
    If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or
        KeyAscii = vbKeyDecPt Or KeyAscii = vbKeyBack Then
        Exit Sub
    Else
        KeyAscii = 0
        Beep
    End If
End Sub
```
End If
End Sub
Private Sub txtMonths_KeyPress (KeyAscii As Integer)
'Only allow number keys, decimal point, or backspace
If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or
KeyAscii = vbKeyDecPt Or KeyAscii = vbKeyBack Then
Exit Sub
Else
KeyAscii = 0
Beep
End If
End Sub

(In the If statements above, note the word processor causes a line break where there really shouldn’t be one. That is, there is no line break between the words Or KeyAscii and = vbKeyDecPt. One appears due to page margins. In all code in these notes, always look for such things.)

3. Rerun the application and test the key trapping performance.

5.2 Select Case - Another Way to Branch

- In addition to If/Then/Else type statements, the Select Case format can be used when there are multiple selection possibilities.

  The Case structure goes as follows:

  ```vba
  Dim x As Integer
  Select Case x
  Case 0
    statement
  Case 1
    statement
  Case Else
    statement
  End Select
  ```

- Say we’ve written this code using the If statement:

  ```vba
  If Age = 5 Then
  Category = "Five Year Old"
  ElseIf Age >= 13 and Age <= 19 Then
  Category = "Teenager"
  ElseIf (Age >= 20 and Age <= 35) Or Age = 50 Or (Age >= 60 and Age <= 65) Then
  Category = "Special Adult"
  ElseIf Age > 65 Then
  Category = "Senior Citizen"
  Else
  Category = "Everyone Else"
  End If
  ```

  The corresponding code with Select Case would be:

  ```vba
  Select Case Age
  ```
Case 5
Category = "Five Year Old"
Case 13 To 19
Category = "Teenager"
Case 20 To 35, 50, 60 To 65
Category = "Special Adult"
Case Is > 65
Category = "Senior Citizen"
Case Else
Category = "Everyone Else"
End Select

Notice there are several formats for the Case statement. Consult on-line help for discussions of these formats.

5.3 The GoTo Statement

- Another branching statement, and perhaps the most hated statement in programming, is the GoTo statement. However, we will need this to do Run-Time error trapping. The format is GoTo Label, where Label is a labeled line. Labeled lines are formed by typing the Label followed by a colon.
- GoTo Example:
  Line10:
  
  GoTo Line10
  When the code reaches the GoTo statement, program control transfers to the line labeled Line10.

6.0 Summary: In this unit we studied about Variables, data types, operators, Constants, functions and branching statements available in Visual Basic. We explain all these things with the help of several examples. We studied how to write statements and expressions in Visual Basic. We saw the scope and lifetime of variables available in Visual basic.
Unit 2: Control flow and Arrays

Unit Contains

1. Introduction
2. Loop Structures
   2.1 While loops
   2.2 Do Loops
   2.3 For Loops
   2.4 For Each Loops
3. Exit Statement
4. End Statement
5. Stop Statement
6. Nested Control Structures
7. Other Control Structures
   7.1 Using Statement
   7.2 With-End With Statement
8. Arrays
   8.1 Fixed-Size Arrays
   8.2 Dynamic Arrays
   8.3 Retrieving the Contents of an Array
   8.4 Adding New Elements on the Fly
   8.5 Erasing an Array
   8.6 Multidimensional Arrays
9. Summary

1. Introduction
Control structures allow you to regulate the flow of your program's execution. Using control structures, you can write Visual Basic code that makes decisions or that repeats actions. Other control structures let you guarantee disposal of a resource or run a series of statements on the same object reference.

2. Loop Structures
Running a set of statements until a condition becomes true

```
<table>
<thead>
<tr>
<th>Previous code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perform actions to be repeated</td>
</tr>
<tr>
<td>Is the condition true?</td>
</tr>
</tbody>
</table>
  No |
  Yes |
| Next code to run |
```
Visual basic mainly supports following looping Structures:

**While Loops**
The **While...End While** construction runs a set of statements as long as the condition specified in the **While** statement is **True**.

**Do Loops**
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 Loops**
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 Each Loops**
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.

### 2.1 While loops
Runs a series of statements as long as a given condition is **True**.

Syntax is:

While condition
    [ statements ]
    [ Exit While ]
    [ statements ]
End While

*condition*
Required. **Boolean** expression. If *condition* is **Nothing**, Visual Basic treats it as **False**.

*statements*
Optional. One or more statements following **While**, which run every time *condition* is **True**.

Exit While
Optional. Transfers control out of the **While** block.

End While
Required. Terminates the definition of the **While** block.
Use a **While...End While** structure when you want to repeat a set of statements an indefinite number of times, as long as a condition remains **True**. If you want more flexibility with where you test the condition or what result you test it for, you might prefer the Do...Loop Statement (Visual Basic). If you want to repeat the statements a set number of times, the For...Next Statement (Visual Basic) is usually a better choice. If *condition* is **True**, all of the *statements* run until the **End While** statement is encountered. Control then returns to the **While** statement and *condition* is again checked. If *condition* is still **True**, the process is repeated. If it is **False**, control passes to the statement following the **End While** statement.

**Rules**

- **Nature of Condition.** The condition usually results from a comparison of two values, but it can be any expression that evaluates to a Boolean Data Type (Visual Basic) value (**True** or **False**). This includes values of other data types, such as numeric types, that have been converted to **Boolean**.

- **Testing the Condition.** The **While** statement always checks the condition before it begins the loop. Looping continues while the condition remains **True**.

- **Number of Iterations.** If *condition* is **False** when you first enter the loop, it does not run even once.

- **Nesting Loops.** You can nest **While** loops by placing one loop within another. You can also nest different kinds of control structures within one another.

- **Transferring Out of the Loop.** The Exit Statement (Visual Basic) transfers control immediately to the statement following the **End While** statement. You might want to exit a loop if you detect a condition that makes it unnecessary or impossible to continue iterating, such as an erroneous value or a termination request. You can place any number of **Exit While** statements anywhere in the **While** loop. **Exit While** is often used after evaluating some condition, for example in an **If...Then...Else** structure.

- **Endless Loops.** One use of **Exit While** is to test for a condition that could cause an *endless loop*, which is a loop that could run an extremely large or even infinite number of times. If you detect such a condition, you can use **Exit While** to escape the loop.

**Example**

This example uses the **While...End While** structure to increment a counter variable. The statements in the loop run as long as the condition evaluates to **True**.

```vba
Dim counter As Integer = 0
While counter < 20
    counter += 1
    ' Insert code to use current value of counter.
End While
MsgBox("While loop ran " & CStr(counter) & " times")
```
2.2 Do Loop
Repeats a block of statements while a Boolean condition is True or until the condition becomes True.
Syntax:
Do { While | Until } condition
   [ statements ]
   [ Exit Do ]
   [ statements ]
Loop
-or-
Do
   [ statements ]
   [ Exit Do ]
   [ statements ]
Loop { While | Until } condition

While
   Required unless Until is used. Repeat the loop until condition is False.

Until
   Required unless While is used. Repeat the loop until condition is True.

condition
   Optional. Boolean expression. If condition is Nothing, Visual Basic treats it as False.

statements
   Optional. One or more statements that are repeated while, or until, condition is True.

Exit Do
   Optional. Transfers control out of the Do loop.

Loop
   Required. Terminates the definition of the Do loop.

Use a Do...Loop structure when you want to repeat a set of statements an indefinite number of times, until a condition is satisfied.
The Do...Loop structure gives you more flexibility than the While...End While Statement (Visual Basic) because it allows you to choose whether to end the loop when condition stops being True or when it first becomes True. It also allows you to test condition at either the beginning or the end of the loop.

Rules
- Nature of Condition. The condition usually results from a comparison of two values, but it can be any expression that evaluates to a Boolean Data Type
(Visual Basic) value (True or False). This includes values of other data types, such as numeric types, that have been converted to Boolean.

- **Testing the Condition.** You can test condition only once, at either the beginning or the end of the loop. You can use either While or Until to specify condition, but not both.

- **Number of Iterations.** If you test condition at the beginning of the loop (in the Do statement), the loop might never run even once. If you test at the end of the loop (in the Loop statement), the loop always runs at least once.

- **Nesting Loops.** You can nest Do loops by placing one loop within another. You can also nest different kinds of control structures within one another.

- **Transferring Out of the Loop.** The Exit Statement (Visual Basic) transfers control immediately to the statement following the Loop statement. You might want to exit a loop if you detect a condition that makes it unnecessary or impossible to continue iterating, such as an erroneous value or a termination request. You can place any number of Exit Do statements anywhere in the Do loop. Exit Do is often used after evaluating some condition, for example in an If...Then...Else structure.

**Endless Loops**

One use of Exit Do is to test for a condition that could cause an endless loop, which is a loop that could run an extremely large or even infinite number of times. If you detect such a condition, you can use Exit Do to escape the loop. Otherwise, the loop continues running.

In the following example, number is assigned a value that could cause the loop to run more than $2^{31}$ times. The If statement checks for this condition and exits if it exists, preventing endless looping.

Example

```vbnet
Sub exitDoExample()
    Dim counter As Integer = 0
    Dim number As Integer = 8
    Do Until number = 10
        If number <= 0 Then Exit Do
        number -= 1
        counter += 1
    Loop
    MsgBox("The loop ran " & counter & " times.")
End Sub
```

To stop an endless loop, Press ESC or CTRL+BREAK.

Example
The following example illustrates nested Do...Loop structures, as well as the use of While and Until, and testing at the beginning (Do statement) and end (Loop statement) of the loop.

```vbnet
Dim check As Boolean = True
Dim counter As Integer = 0
Do
    Do While counter < 20
        counter += 1
        If counter = 10 Then
            check = False
            Exit Do
        End If
    Loop
    Loop Until check = False
End Sub
```

In the preceding example, the inner Do...Loop structure loops 10 times, sets the value of the flag to False, and exits prematurely using the Exit Do statement. The outer loop exits immediately upon checking the value of the flag.

2.3 For loops
Repeats a group of statements a specified number of times.

```
For counter [ As datatype ] = start To end [ Step step ]
[ statements ]
[ Exit For ]
[ statements ]
Next [ counter ]
```

- **counter**
  Required in the For statement. Numeric variable. The control variable for the loop.

- **datatype**
  Required if counter is not already declared. Data type of counter.

- **start**
  Required. Numeric expression. The initial value of counter.

- **end**
  Required. Numeric expression. The final value of counter.

- **step**
  Optional. Numeric expression. The amount by which counter is incremented each time through the loop.

- **statements**
Optional. One or more statements between **For** and **Next** that run the specified number of times.

**Exit For**
Optional. Transfers control out of the **For** loop.

**Next**
Required. Terminates the definition of the **For** loop.

Use a **For...Next** structure when you want to repeat a set of statements a set number of times.
A **While...End While** Statement (Visual Basic) or **Do...Loop** Statement (Visual Basic) works well when you do not know in advance how many times you need to run the statements in the loop. However, when you expect to run the loop a specific number of times, a **For...Next** loop is a better choice. You determine the number of iterations when you first enter the loop. The value of **step** can be either positive or negative. It determines loop processing as follows:

<table>
<thead>
<tr>
<th>Step Value</th>
<th>Loop executes if</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive or zero</td>
<td>counter &lt;= end</td>
</tr>
<tr>
<td>Negative</td>
<td>counter &gt;= end</td>
</tr>
</tbody>
</table>

If not specified, **step** defaults to 1.

**Rules**

- **Data Types.** The data type of **counter** is usually **Integer** but can be any type that supports the greater than or equal to (**>=**), less than or equal to (**<=**), addition (**+**), and subtraction (**-**) operators. It can even be a user-defined type provided it supports all these operators.
  The **start**, **end**, and **step** expressions usually evaluate to type **Integer** but can evaluate to any data type that widens to the type of **counter**. If you use a user-defined type for **counter**, this means you might have to define the **CType** conversion operator to convert the types of **start**, **end**, or **step** to the type of **counter**.
- **Declaration.** If **counter** has not been declared outside this loop, you must declare it within the **For** statement. In this case, the scope of **counter** is the body of the loop. However, you cannot declare **counter** both outside and inside the loop.
- **Number of Iterations.** Visual Basic evaluates the iteration values **start**, **end**, and **step** only once, before the loop begins. If your statement block changes **end** or **step**, these changes do not affect the iteration of the loop.
- **Nesting Loops.** You can nest **For** loops by placing one loop within another. However, each loop must have a unique **counter** variable. The following construction is valid.
  
  For **i** As **Integer** = 1 To 10
For j As Integer = 1 To 10
    For k As Integer = 1 To 10
        ' Insert statements to operate with current values of i, j, and k.
        Next k
    Next j
Next i

You can also nest different kinds control structures within one another. If a `Next` statement of an outer nesting level is encountered before the `Next` of an inner level, the compiler signals an error. However, the compiler can detect this overlapping error only if you specify `counter` in every `Next` statement.

- **Identifying the Control Variable.** You can optionally specify `counter` in the `Next` statement. This improves the readability of your program, especially if you have nested `For` loops. You must specify the same variable as the one that appears in the corresponding `For` statement.

- **Transferring Out of the Loop.** The Exit Statement (Visual Basic) transfers control immediately to the statement following the `Next` statement. You might want to exit a loop if you detect a condition that makes it unnecessary or impossible to continue iterating, such as an erroneous value or a termination request. Also, if you catch an exception in a `Try...Catch...Finally`, you can use `Exit For` at the end of the `Finally` block.

You can place any number of `Exit For` statements anywhere in the `For` loop. `Exit For` is often used after evaluating some condition, for example in an `If...Then...Else` structure.

- **Endless Loops.** One use of `Exit For` is to test for a condition that could cause an *endless loop*, which is a loop that could run an extremely large or even infinite number of times. If you detect such a condition, you can use `Exit For` to escape the loop.

**Behavior**

- **Entry into the Loop.** When execution of the `For...Next` loop begins, Visual Basic evaluates `start`, `end`, and `step` for the only time. It then assigns `start` to `counter`. Before it runs the statement block, it compares `counter` to `end`. If `counter` is already past the end value, the `For` loop terminates and control passes to the statement following the `Next` statement. Otherwise the statement block runs.

- **Iterations of the Loop.** Each time Visual Basic encounters the `Next` statement, it increments `counter` by `step` and returns to the `For` statement. Again it compares `counter` to `end`, and again it either runs the block or terminates the loop depending on the result. This process continues until `counter` passes `end` or an `Exit For` statement is encountered.
Termination of the Loop. The loop does not terminate until \textit{counter} has passed \textit{end}. If \textit{counter} is equal to \textit{end}, the loop continues. The comparison that determines whether to run the block is \textit{counter} <= \textit{end} if \textit{step} is positive and \textit{counter} >= \textit{end} if \textit{step} is negative.

Changing Iteration Values. Changing the value of \textit{counter} while inside a loop can make it more difficult to read and debug your code. Changing the value of \textit{start}, \textit{end}, or \textit{step} does not affect the iteration values determined when the loop was first entered.

Example

The following example demonstrates nested \textbf{For...Next} structures with different step values.

Dim words, digit As Integer
Dim thisString As String = ""
For words = 10 To 1 Step -1
    For digit = 0 To 9
        thisString &= CStr(digit)
    Next digit
    thisString &= " \\
Next words

The preceding example creates a string that contains 10 instances of the numbers 0 through 9, each string separated from the other by a single space. The outer loop decrements a loop counter variable each time through the loop.

2.4 For Each loops

Repeats a group of statements for each element in a collection.

Syntax:
For Each element [ As datatype ] In group
    [ statements ]
    [ Exit For ]
    [ statements ]
Next [ element ]

\textit{element} \\
Required in the \textbf{For Each} statement. Optional in the \textbf{Next} statement. Variable. Used to iterate through the elements of the collection.

\textit{datatype} \\
Required if \textit{element} is not already declared. Data type of \textit{element}.

\textit{group} \\
Required. Object variable. Refers to the collection over which the \textit{statements} are to be repeated.
Optional. One or more statements between **For Each** and **Next** that run on each item in *group*.

**Exit For**

Optional. Transfers control out of the **For Each** loop.

**Next**

Required. Terminates the definition of the **For Each** loop.

Use a **For Each...Next** loop when you want to repeat a set of statements for each element of a collection or array.

A For...Next Statement (Visual Basic) works well when you can associate each iteration of a loop with a control variable and determine that variable's initial and final values. However, when you are dealing with a collection, the concept of initial and final values is not meaningful, and you do not necessarily know how many elements the collection has. In this case, a **For Each...Next** loop is a better choice.

**Rules**

- **Data Types.** The data type of *element* must be such that the data type of the elements of *group* can be converted to it.

  The data type of *group* must be a reference type that refers to a collection or an array. This means that *group* must refer to an object that implements the `IEnumerable` interface of the `System.Collections` namespace or the `IEnumerable` interface of the `System.Collections.Generic` namespace.

  `IEnumerable` defines the GetEnumerator method, which returns an enumerator object for the collection. The enumerator object implements the `IEnumerator` interface of the `System.Collections` namespace and exposes the Current property and the Reset and MoveNext methods. Visual Basic uses these to traverse the collection.

  The elements of *group* are usually of type `Object` but can have any run-time data type.

- **Declaration.** If *element* has not been declared outside this loop, you must declare it within the **For Each** statement. In this case, the scope of *element* is the body of the loop. However, you cannot declare *element* both outside and inside the loop.

- **Number of Iterations.** Visual Basic evaluates the collection only once, before the loop begins. If your statement block changes *element* or *group*, these changes do not affect the iteration of the loop.

- **Nesting Loops.** You can nest **For Each** loops by placing one loop within another. However, each loop must have a unique *element* variable.

  You can also nest different kinds of control structures within one another.

  If a **Next** statement of an outer nesting level is encountered before the **Next** of an inner level, the compiler signals an error. However, the compiler can detect this overlapping error only if you specify *element* in every **Next** statement.
- **Identifying the Control Variable.** You can optionally specify `element` in the `Next` statement. This improves the readability of your program, especially if you have nested `For Each` loops. You must specify the same variable as the one that appears in the corresponding `For Each` statement.

- **Transferring Out of the Loop.** The Exit Statement (Visual Basic) transfers control immediately to the statement following the `Next` statement. You might want to exit a loop if you detect a condition that makes it unnecessary or impossible to continue iterating, such as an erroneous value or a termination request. Also, if you catch an exception in a `Try...Catch...Finally` statement, you can use `Exit For` at the end of the ` Finally` block. You can place any number of `Exit For` statements anywhere in the `For Each` loop. `Exit For` is often used after evaluating some condition, for example in an `If...Then...Else` structure.

- **Endless Loops.** One use of `Exit For` is to test for a condition that could cause an *endless loop*, which is a loop that could run an extremely large or even infinite number of times. If you detect such a condition, you can use `Exit For` to escape the loop.

**Behavior**

- **Entry into the Loop.** When execution of the `For Each...Next` loop begins, Visual Basic verifies that `group` refers to a valid collection object. If not, it throws an exception. Otherwise, it calls the `MoveNext` method and the `Current` property of the enumerator object to return the first element. If `MoveNext` indicates that there is no next element, that is, if the collection is empty, then the `For Each` loop terminates and control passes to the statement following the `Next` statement. Otherwise, Visual Basic sets `element` to the first element and runs the statement block.

- **Iterations of the Loop.** Each time Visual Basic encounters the `Next` statement, it returns to the `For Each` statement. Again it calls `MoveNext` and `Current` to return the next element, and again it either runs the block or terminates the loop depending on the result. This process continues until `MoveNext` indicates that there is no next element or an `Exit For` statement is encountered.

- **Termination of the Loop.** When all the elements in the collection have been successively assigned to `element`, the `For Each` loop terminates and control passes to the statement following the `Next` statement.

- **Changing Iteration Values.** Changing the value of `element` while inside a loop can make it more difficult to read and debug your code. Changing the value of `group` does not affect the collection or its elements, which were determined when the loop was first entered.
- **Traversal Order.** When you execute a **For Each**...**Next** loop, traversal of the collection is under the control of the enumerator object returned by the **GetEnumerator** method. The order of traversal is not determined by Visual Basic, but rather by the **MoveNext** method of the enumerator object. This means that you might not be able to predict which element of the collection is the first to be returned in an element, or which is the next to be returned after a given element.

If your code depends on traversing a collection in a particular order, a **For Each**...**Next** loop is not the best choice unless you know the characteristics of the enumerator object the collection exposes. You might achieve more reliable results using a different loop structure, such as **For**...**Next** or **Do**...**Loop**.

- **Modifying the Collection.** The enumerator object returned by **GetEnumerator** normally does not allow you to alter the collection by adding, deleting, replacing, or reordering any elements. If you alter the collection after you have initiated a **For Each**...**Next** loop, the enumerator object becomes invalid, and the next attempt to access an element results in an **InvalidOperationException** exception.

However, this blocking of modification is not determined by Visual Basic, but rather by the implementation of the **IEnumerable** interface. It is possible to implement **IEnumerable** in such a way as to allow modification during iteration. If you are considering doing such dynamic modification, be sure you understand the characteristics of the **IEnumerable** implementation on the collection you are using.

- **Modifying Collection Elements.** The **Current** property of the enumerator object is **ReadOnly** (Visual Basic), and it returns a local copy of each collection element. This means that you cannot modify the elements themselves in a **For Each**...**Next** loop. Any modification you make affects only the local copy from **Current** and is not reflected back into the underlying collection. However, if an element is a reference type, you can modify the members of the instance to which it points. The following example illustrates this.

**Example**

```vba
Sub lightBlueBackground(ByVal thisForm As System.Windows.Forms.Form)
    For Each thisControl As System.Windows.Forms.Control In thisForm.Controls
        thisControl.BackColor = System.Drawing.Color.LightBlue
        Next thisControl
    End Sub
```

The preceding example can modify the **BackColor** member of each **thisControl** element, although it cannot modify **thisControl** itself.
- **Traversing Arrays.** Because the Array class implements the `IEnumerable` interface, all arrays expose the GetEnumerator method. This means that you can iterate through an array with a `For Each...Next` loop. However, you can only read the array elements, not change them. For an illustration, see How to: Run Several Statements for Each Element in a Collection or Array.

**Example**
The following example uses the `For Each...Next` statement to search all elements in a collection for the string "Hello". The example assumes that the collection `thisCollection` has already been created and that its elements are of type `String`.

```vbnet
Dim found As Boolean = False
Dim thisCollection As New Collection
For Each thisObject As String In thisCollection
    If thisObject = "Hello" Then
        found = True
        Exit For
    End If
Next thisObject
```

### 3. Exit Statement
Exits a procedure or block and transfers control immediately to the statement following the procedure call or the block definition.

```vbnet
Exit { Do | For | Function | Property | Select | Sub | Try | While }
```

**Do**
Immediately exits the `Do` loop in which it appears. Execution continues with the statement following the `Loop` statement. `Exit Do` can be used only inside a `Do` loop. When used within nested `Do` loops, `Exit Do` exits the innermost loop and transfers control to the next higher level of nesting.

**For**
Immediately exits the `For` loop in which it appears. Execution continues with the statement following the `Next` statement. `Exit For` can be used only inside a `For...Next` or `For Each...Next` loop. When used within nested `For` loops, `Exit For` exits the innermost loop and transfers control to the next higher level of nesting.

**Function**
Immediately exits the `Function` procedure in which it appears. Execution continues with the statement following the statement that called the `Function` procedure. `Exit Function` can be used only inside a `Function` procedure.

**Property**
Immediately exits the `Property` procedure in which it appears. Execution continues with the statement that called the `Property` procedure, that is, with
the statement requesting or setting the property's value. **Exit Property** can be used only inside a property's **Get** or **Set** procedure.

**Select**
Immediately exits the **Select Case** block in which it appears. Execution continues with the statement following the **End Select** statement. **Exit Select** can be used only inside a **Select Case** statement.

**Sub**
Immediately exits the **Sub** procedure in which it appears. Execution continues with the statement following the statement that called the **Sub** procedure. **Exit Sub** can be used only inside a **Sub** procedure.

**Try**
Immediately exits the **Try** or **Catch** block in which it appears. Execution continues with the **Finally** block if there is one, or with the statement following the **End Try** statement otherwise. **Exit Try** can be used only inside a **Try** or **Catch** block, and not inside a **Finally** block.

**While**
Immediately exits the **While** loop in which it appears. Execution continues with the statement following the **End While** statement. **Exit While** can be used only inside a **While** loop. When used within nested **While** loops, **Exit While** transfers control to the loop that is one nested level above the loop where **Exit While** occurs.

Do not confuse **Exit** statements with **End** statements. **Exit** does not define the end of a statement.

**Example**
The following example uses the **Exit** statement to exit a **For...Next** loop, a **Do** loop, and a **Sub** procedure.

```vbnet
Sub exitStatementDemo()
    Dim demoNum As Single
    ' Set up an infinite loop.
    Do
        For i As Integer = 1 To 10000000
            demoNum = Int(Rnd() * 100)
            Select Case demoNum
                Case 7 : Exit For
                Case 29 : Exit Do
                Case 54 : Exit Sub
            End Select
        Next i
    Loop
```

87
4. End Statement
Terminates execution immediately.
You can place the End statement anywhere in a procedure to force the entire application to stop running. End closes any files opened with an Open statement and clears all the application's variables. The application closes as soon as there are no other programs holding references to its objects and none of its code is running.
The End statement stops code execution abruptly, without invoking the Dispose or Finalize method, or any other Visual Basic code. Object references held by other programs are invalidated. If an End statement is encountered within a Try or Catch block, control does not pass to the corresponding Finally block. The End statement stops code execution abruptly, without invoking the Dispose or Finalize method, or any other Visual Basic code. Object references held by other programs are invalidated. If an End statement is encountered within a Try or Catch block, control does not pass to the corresponding Finally block.
The Stop statement suspends execution, but unlike End, it does not close any files or clear any variables, unless it is encountered in a compiled executable (.exe) file.
Because End terminates your application without attending to any resources that might be open, you should try to close down cleanly before using it. For example, if your application has any forms open, you should close them before control reaches the End statement.
You should use End sparingly, and only when you need to stop immediately. The normal ways to terminate a procedure (Return Statement (Visual Basic) and Exit Statement (Visual Basic)) not only close down the procedure cleanly but also give the calling code the opportunity to close down cleanly. A console application, for example, can simply Return from the Main procedure.
The End statement calls the Exit method of the Environment class in the System namespace. Exit requires that you have UnmanagedCode permission. If you do not, a SecurityException error occurs.
When followed by an additional keyword, End (Visual Basic) delineates the end of the definition of the appropriate procedure or block. For example, End Function terminates the definition of a Function procedure.

Example
The following example uses the End statement to terminate code execution if the user requests it.

Sub Form_Load()
    Dim answer As MsgBoxResult
    answer = MsgBox("Do you want to quit now?", MsgBoxStyle.YesNo)
End Sub
If answer = MsgBoxResult.Yes Then
    MsgBox("Terminating program")
End
End If
End Sub

5. Stop Statement
Suspends execution.
You can place **Stop** statements anywhere in procedures to suspend execution. Using the **Stop** statement is similar to setting a breakpoint in the code.
The **Stop** statement suspends execution, but unlike **End**, it does not close any files or clear any variables, unless it is encountered in a compiled executable (.exe) file.
If the **Stop** statement is encountered in code that is running outside of the integrated development environment (IDE), the debugger is invoked. This is true regardless of whether the code was compiled in debug or retail mode.

**Example**
This example uses the **Stop** statement to suspend execution for each iteration through the **For...Next** loop.

Dim i As Integer
For i = 1 To 10
    Debug.WriteLine(i)
    ' Stop during each iteration and wait for user to resume.
    Stop
Next i

6. Nested Control Structures
You can place control statements inside other control statements, for example an **If...Then...Else** block within a **For...Next** loop. A control statement placed inside another control statement is said to be **nested**.

**Nesting Levels**
Control structures in Visual Basic can be nested to as many levels as you want. It is common practice to make nested structures more readable by indenting the body of each one. The integrated development environment (IDE) editor automatically does this.
In the following example, the procedure **sumRows** adds together the positive elements of each row of the matrix.

Public Sub sumRows(ByVal a(,) As Double, ByRef r() As Double)
    Dim i, j As Integer
    For i = 0 To UBound(a, 1)
        r(i) = 0
    Next i
End Sub
For j = 0 To UBound(a, 2)
    If a(i, j) > 0 Then
        r(i) = r(i) + a(i, j)
    End If
Next j
Next i
End Sub

In the preceding example, the first **Next** statement closes the inner **For** loop and the last **Next** statement closes the outer **For** loop.
Likewise, in nested **If** statements, the **End If** statements automatically apply to the nearest prior **If** statement. Nested **Do** loops work in a similar fashion, with the innermost **Loop** statement matching the innermost **Do** statement.

**Nesting Different Kinds of Control Structures**
You can nest one kind of control structure within another kind. The following example uses a **With** block inside a **For Each** loop and nested **If** blocks inside the **With** block.

For Each ctl As System.Windows.Forms.Control In Me.Controls
    With ctl
        If .CanFocus Then
            .Text = "Colors changed"
            If Not .Focus() Then
                ' Insert code to process failed focus.
                End If
        End If
    End With
Next ctl

**Overlapping Control Structures**
You cannot overlap control structures. This means that any nested structure must be completely contained within the next innermost structure. For example, the following arrangement is invalid because the **For** loop terminates before the inner **With** block terminates.

**Invalid nesting of For and With structures**
The Visual Basic compiler detects such overlapping control structures and signals a compile-time error.

7. Other Control Structures
Visual Basic provides control structures that help you dispose of a resource or reduce the number of times you have to repeat an object reference.

**Using...End Using Construction**
The *Using...End Using* construction establishes a statement block within which you make use of a resource such as a SQL connection. You can optionally acquire the resource with the *Using* statement. When you exit the *Using* block, Visual Basic automatically disposes of the resource so that it is available for other code to use. The resource must be local and disposable.

**With...End With Construction**
The *With...End With* construction allows you to specify an object reference once and then run a series of statements that access its members. This can simplify your code and improve performance because Visual Basic does not have to re-establish the reference for each statement that accesses it.

### 7.1 Using Statement
Declares the beginning of a *Using* block and optionally acquires the system resources that the block controls.

**Syntax:**
```
Using { resourcelist | resourceexpression }
  [ statements ]
End Using
```

*resourcelist*

Required if you do not supply *resourceexpression*. List of one or more system resources that this *Using* block controls.

*resourceexpression*

Required if you do not supply *resourcelist*. Reference variable or expression referring to a system resource to be controlled by this *Using* block.

*statements*

Optional. Block of statements that the *Using* block runs.
End Using
   Required. Terminates the definition of the Using block and disposes of all the
   resources that it controls.

Each resource in the resourcelist part has the following syntax and parts:
resourcename As New resourcetype [ ( [ arglist ] ) ]
-or-
resourcename As resourcetype = resourceexpression

resourcename
   Required. Reference variable that refers to a system resource that the Using
   block controls.

New
   Required if the Using statement acquires the resource. If you have already
   acquired the resource, use the second syntax alternative.

resourcetype
   Required. The class of the resource. The class must implement the
   IDisposable interface.

arglist
   Optional. List of arguments you are passing to the constructor to create an
   instance of resourcetype..

resourceexpression
   Required. Variable or expression referring to a system resource satisfying the
   requirements of resourcetype. If you use the second syntax alternative, you
   must acquire the resource before passing control to the Using statement.

Sometimes your code requires an unmanaged resource, such as a file handle, a COM
wrapper, or a SQL connection. A Using block guarantees the disposal of one or more
such resources when your code is finished with them. This makes them available for
other code to use.

Managed resources are disposed of by the .NET Framework garbage collector (GC)
without any extra coding on your part. You do not need a Using block for managed
resources.

A Using block has three parts: acquisition, usage, and disposal.

- Acquisition means creating a variable and initializing it to refer to the system
  resource. The Using statement can acquire one or more resources, or you
  can acquire exactly one resource before entering the block and supply it to the
  Using statement. If you supply resourceexpression, you must acquire the
  resource before passing control to the Using statement.
- **Usage** means accessing the resources and performing actions with them. The statements between **Using** and **End Using** represent the usage of the resources.

- **Disposal** means calling the **Dispose** method on the object in `resourcename`. This allows the object to cleanly terminate its unmanaged resource. The **End Using** statement disposes of the resources under the **Using** block's control.

**Behavior**
A **Using** block behaves like a **Try...Finally** construction in which the **Try** block uses the resources and the **Finally** block disposes of them. Because of this, the **Using** block guarantees disposal of the resources, no matter how you exit the block. This is true even in the case of an unhandled exception, except for a **StackOverflowException**. The scope of every resource variable acquired by the **Using** statement is limited to the **Using** block.
If you specify more than one system resource in the **Using** statement, the effect is the same as if you nested **Using** blocks one within another.

**Structured Exception Handling Within a Using Block**
If you need to handle an exception that might occur within the **Using** block, you can add a complete **Try...Finally** construction to it. If you need to handle the case where the **Using** statement is not successful in acquiring a resource, you can test to see if `resourcename` is **Nothing**.

**Structured Exception Handling Instead of a Using Block**
If you need finer control over the acquisition of the resources, or you need additional code in the **Finally** block, you can rewrite the **Using** block as a **Try...Finally** construction. The following example shows skeleton **Try** and **Using** constructions that are equivalent in the acquisition and disposal of resource.
Using resource As New resourceType
  ' Insert code to work with resource.
End Using
  ' THE FOLLOWING TRY CONSTRUCTION IS EQUIVALENT TO THE USING BLOCK
Dim resource As New resourceType
Try
  ' Insert code to work with resource.
Catch ex As Exception
  ' Insert code to process exception.
Finally
  ' Insert code to do additional processing before disposing of resource.
  resource.Dispose()
End Try
The code inside the **Using** block should not assign the object in `resourcename` to another variable. When you exit the **Using** block, the resource is disposed, and the other variable cannot access the resource to which it points.

**Example**
The following example uses a **Using** block to acquire a new font. This guarantees that the system calls the **Dispose** method on the font when your code exits the block.

```vbnet
Public Sub setbigbold(ByVal c As Control)
    c.Font = nf
    c.Text = "This is 12-point Arial bold"
  End Using
End Sub
```

### 7.2 With...End With Statement
Executes a series of statements making repeated reference to a single object or structure.

```vbnet
With object
  [ statements ]
End With
```

- **object**
  Required. Variable or expression. Can evaluate to any data type, including elementary types.

- **statements**
  Optional. One or more statements between **With** and **End With** that run on **object**.

**End With**
Required. Terminates the definition of the **With** block.

**With...End With** allows you to perform a series of statements on a specified object without requalifying the name of the object. If the qualification path to the object is long, using **With...End With** can improve your performance. A **With** block also reduces repetitive typing of the qualification path and the risk of mistyping one of its elements.

For example, to change a number of different properties on a single object, place the property assignment statements inside the **With...End With**, referring to the object only once instead of for each property assignment.
Rules

- **Data Types.** The data type of *object* can be any class or structure type, or even a Visual Basic elementary type such as **Integer**. The .NET Framework supports all elementary types with classes or structures, which have members you can access inside a **With** block.

- **Declaration.** You must declare *object* before you enter the **With** block. You cannot declare it in the **With** statement.

- **Number of Iterations.** A **With** block is not an iterative construction. Unless there is a loop inside the block, the statements run only once.

- **Nesting Structures.** You can nest **With**...**End With** structures by placing one structure within another.

  However, because members of outer statements are masked inside the inner statements, you must provide a fully qualified object reference in an inner **With** block to any member of an object in an outer **With** block.

You can also nest different kinds of control structures within one another.

**Transferring Out of the Structure.** Visual Basic does not support the Exit Statement (Visual Basic) to transfer control out of a **With** block. If you need to exit before all the statements have been executed, put a label on the **End With** statement and use the GoTo Statement to branch to it.

You cannot transfer control either from outside a **With** block to inside it, or from inside it to the outside. You can call a procedure from inside the block, but control returns to the following statement.

- **Accessing Other Objects.** Once you have entered a **With** block, you cannot reassign *object* until you have passed the **End With** statement. Therefore, you can access the methods and properties of only the specified object without qualifying them. You can use methods and properties of other objects, but you must qualify them with their object names.

**Example**

The following example uses a **With** block to execute a series of statements on a single object. The example assumes that object testObject has already been created and exposes the referenced properties.

```
With testObject
  .Height = 100
  .Text = "Hello, World"
End With
```
Transfer Control Out of a Control Structure

You can use the Exit Statement (Visual Basic) to exit directly from a control structure. Exit immediately transfers execution to the statement following the last statement of the control structure. The syntax of the Exit statement specifies which type of control structure you are transferring out of. The following versions of the Exit statement are possible:

- Exit Select
- Exit Try
- Exit While
- Exit Do
- Exit For

Exit statements can appear as many times as needed within a control structure that supports them. Exit is useful when a control structure has done everything it needs to do and does not need to run any more statements.

Control Structures that Do Not Support Exit.

You cannot use the Exit statement to transfer out of an If, Using, or With block. To achieve the same result, you can put a statement label on the block’s End statement and transfer to it using a GoTo statement.

Example

If an Exit statement is encountered within nested control structures, control passes to the statement following the end of the innermost structure of the kind specified in the Exit statement. The following example illustrates this.

```vba
Public Sub invertElements(ByRef a(,) As Double)
    For i As Integer = 0 To UBound(a, 1)
        For j As Integer = 0 To UBound(a, 2)
            If a(i, j) = 0 Then
                ' Cannot complete this row; resume outer loop.
                Exit For
            Else
                a(i, j) = 1.0 / a(i, j)
            End If
        Next j
        ' Control comes here directly from the Exit For statement.
    Next i
End Sub
```

In the preceding example, the Exit For statement is located in the inner For loop, so it passes control to the statement following that loop and continues with the outer For loop.

8. Arrays in Visual Basic

An array is a set of values that are logically related to each other, such as the number of students in each grade in a grammar school.
An array allows you to refer to these related values by the same name and to use a number, called an index or subscript, to tell them apart. The individual values are called the elements of the array. They are contiguous from index 0 through the highest index value.

Example

The following example declares an array variable to hold the number of students in each grade in a grammar school.

Dim students(6) As Integer

The array students in the preceding example contains 7 elements. The indexes of the elements range from 0 through 6. Having this array is simpler than declaring 7 different variables.

The following illustration shows the array students. For each element of the array:

- The index of the element represents the grade (index 0 represents kindergarten).
- The value contained in the element represents the number of students in that grade.

Elements of the "students" array

The following example shows how to refer to the first, second, and last element of the array students.

Dim kindergarten As Integer = students(0)
Dim firstGrade As Integer = students(1)
Dim sixthGrade As Integer = students(6)
MsgBox(“Students in kindergarten = ” & CStr(kindergarten))
MsgBox(“Students in first grade = ” & CStr(firstGrade))
MsgBox(“Students in sixth grade = ” & CStr(sixthGrade))

You can refer to the array as a whole by using just the array variable name without indexes.
Array Types and Other Types

Data Types

Every array has a data type, but it is not the same as the data type of its elements. For example, the array students in the preceding example is of type \texttt{Integer()}, while each of its elements is of type \texttt{Integer}. The notation \texttt{Integer()} means an array of \texttt{Integer} elements.

Every array inherits from the \texttt{System.Array} class, and you can declare a variable to be of type \texttt{Array}, but you cannot create an array of type \texttt{Array}. Also, the ReDim Statement (Visual Basic) cannot operate on a variable declared as type \texttt{Array}. For these reasons, and for type safety, it is advisable to declare every array as a specific type, such as \texttt{Integer} in the preceding example.

Array Dimensions

The array students in the preceding example uses one index and is said to be one-dimensional. An array that uses more than one index or subscript is called multidimensional.

Another kind of array is one which holds other arrays as elements. This is known as an array of arrays or a jagged array. A jagged array can be either one-dimensional or multidimensional, and so can its elements.

Types of Arrays

An array is a lot like a CD rack. You know: one of those rectangular boxes with slots to slide CDs in, each above another. There are two types of Visual Basic arrays: fixed-size and dynamic.

8.1 Fixed-Size Arrays

A fixed-size array most closely matches our CD rack analogy. There are a limited number of slots you can slide CDs into. Pretend you have three CDs - one by the Deftones, another by Tool, and a third by Disturbed. To fit all of these in your rack, the rack must contain at least three slots. So you declare your CD rack as having three slots:

\begin{verbatim}
Dim strCDRack(0 to 2) As String
\end{verbatim}

You've just made a variable 'strCDRack' that contains three slots (#0, #1, and #2) and is of a String data type. Now you can insert your CDs into it:

\begin{verbatim}
Dim strCDRack(0 to 2) As String
strCDRack(0) = "Deftones"
strCDRack(1) = "Tool"
strCDRack(2) = "Disturbed"
\end{verbatim}

Notice that each of the three new lines starts off with the variable name and then gives an element number before having a value assigned. This is like numbering the slots on your CD rack starting at 0 up to 2 and then inserting a CD into each slot.

The format for declaring an array is:

\begin{verbatim}
Dim|Public|Private ArrayName(Subscript) As DataType
\end{verbatim}
- Dim, Public, and Private declare the array and its scope. Using Dim in a procedure will make the array only available from within that procedure. Using it in the General Declarations section will make it available to all procedures in that module. Private has the same effect and should be used only at the modular level. Using Public will make the array available throughout the project.

- **ArrayName** is the name of the array.

- **Subscript** is the dimensions of the array.

- **DataType** is any valid data type.

### 8.2 Dynamic Arrays

The new Charlotte Church CD came out but your rack only has three slots. You don't want to throw away any of your CDs to make room for the new one so you decide to use your ultimate building skills to attach another slot. You start building:

```vbnet
Dim strCDRack() As String
ReDim strCDRack(0 to 2) As String
```

`strCDRack(0) = "Deftones"
strCDRack(1) = "Tool"
strCDRack(2) = "Disturbed"

What have you done? Nothing wrong, you've just dimensioned your array another way that allows for expansion. Notice that the subscript of the Dim statement is missing. This is OK; it tells VB that your array is a dynamic array, meaning that you can change its size with `ReDim`.

Now that you've rebuilt the structure of your CD rack, allowing for expansion, it is time to expand:

```vbnet
Dim strCDRack() As String
```

```vbnet
ReDim strCDRack(0 to 2) As String
```

```vbnet
strCDRack(0) = "Deftones"
strCDRack(1) = "Tool"
strCDRack(2) = "Disturbed"
```

```vbnet
ReDim Preserve strCDRack(0 to 3) As String
```

```vbnet
strCDRack(3) = "Charlotte Church"
```

This snippet has two more lines, the first redimensioning the array one element larger and the second setting this element's value. Notice the Preserve keyword: it forces Visual Basic to retain all existing elements' values. Without this keyword all your old CDs would be lost and you'd be stuck with just Charlotte Church.

The `ReDim` keyword's syntax is:

```vbnet
ReDim [Preserve] ArrayName(Subscript) As DataType
```

- `ReDim` is the keyword that denotes we are redimensioning an array.

- `Preserve` is an optional keyword that forces Visual Basic to retain all existing elements' values. Without it all elements will return to their default values. (Numeric
data types to 0, variable-length strings to "," (a zero-length string), fixed-length strings filled with zeros, and variants to empty.)

- **ArrayName** is the name of the array.

- **Subscript** is the dimensions of the array.

- **DataType** is any valid data type. The data type cannot be changed from its initial declaration when using the ReDim keyword. (Unless it was initially declared as a Variant.)

### 8.3 Retrieving the Contents of an Array

Now that you know how to build an array, you might ask how to retrieve its contents. Say you've built an array of your friends' names:

```vba
Dim strFriends(0 to 6) As String
strFriends(0) = "Bianca"
strFriends(1) = "Jeana"
strFriends(2) = "Sam"
strFriends(3) = "Jenna"
strFriends(4) = "Erin"
strFriends(5) = "Carolyn"
strFriends(6) = "Kate"
```

That's all good and dandy but you want to display their names in successive message boxes, so you construct a loop:

```vba
Dim strFriends(0 to 6) As String, lngPosition as Long
strFriends(0) = "Bianca"
strFriends(1) = "Jeana"
strFriends(2) = "Sam"
strFriends(3) = "Jenna"
strFriends(4) = "Erin"
strFriends(5) = "Carolyn"
strFriends(6) = "Kate"
For lngPosition = LBound(strFriends) To UBound(strFriends)
    MsgBox strFriends(lngPosition)
Next lngPosition
```

There are two new functions in that snippet of code. **LBound** and **UBound** are used to determine the lower and upper bounds of an array. Because `strFriends` has a lower bound of 0 and an upper bound of 6. These functions allow you to iterate through an array with a dynamic size and they keep you from having to keep track of the array's size yourself. With each iteration of that loop, `lngPosition` will count up from 0 to 6. By accessing the array as `strFriends(lngPosition)` you are greatly reducing the amount of code you have to write.

### 8.4 Adding New Elements on the Fly

Sometimes you have an array that needs to keep growing, and you don't know what the upper bound will end up being. Maybe you are making a crappy MP3 player and need to ask the user to input song names. You might do something like this:

```vba
Dim strSongNames() As String    'Array of song names
Dim blDimensioned As Boolean    'Is the array dimensioned?
Dim strText As String           'To temporarily hold names
```
Dim lngPosition as Long  'Counting

'The array has not yet been dimensioned:
blDimensioned = False

Do

'Ask for a song name
strText = InputBox("Enter a song name:")

If strText <> "" Then

'Has the array been dimensioned?
If blDimensioned = True Then

'Yes, so extend the array one element large than its current upper bound.
'Without the "Preserve" keyword below, the previous elements in our array
would be erased with the resizing
ReDim Preserve strSongNames(0 To UBound(strSongNames) + 1) As String

Else

'No, so dimension it and flag it as dimensioned.
ReDim strSongNames(0 To 0) As String
blDimensioned = True

End If

'Add the song name to the last element in the array.
strSongNames(UBound(strSongNames)) = strText

End If

Loop Until strText = ""

'Display entered song names:
For lngPosition = LBound(strSongNames) To UBound(strSongNames)

    MsgBox strSongNames(lngPosition)

Next lngPosition

'Erase array
Erase strSongName

Look to the comments for an explanation of what is going on.

8.5 Erasing an Array
You should always erase your array when you are done using it, especially if you are
using dynamic arrays. It's rather easy:
Dim strFriends(0 to 2) As String

strFriends(0) = "Bianca"
strFriends(1) = "Jeana"
strFriends(2) = "Erin"
8.6 Multidimensional Arrays
So far all of the examples we've looked at have used one dimensional arrays, but arrays can be multidimensional too. Multidimensional arrays can be thought of as arrays-of-arrays. For example, to visualize a two dimensional array we could picture a row of CD racks. To make things easier, we can imagine that each CD rack could be for a different artist. Like the CDs, the racks would be identifiable by number. Below we'll define a two dimensional array representing a row of CD racks. The strings inside of the array will represent album titles.

' Here we will define an array where the first dimension contains 2 elements and
two second dimension contains 4 elements
ReDim cdRack(0 to 1, 0 to 3) As String

' A CD rack for the Beatles
cdRack(0, 0) = "Rubber Soul"
cdRack(0, 1) = "Revolver"
cdRack(0, 2) = "The White Album"
cdRack(0, 3) = "Let It Be"

' A CD rack for the Rolling Stones
cdRack(1, 0) = "Sticky Fingers"
cdRack(1, 1) = "Beggars Banquet"
cdRack(1, 2) = "Let It Bleed"
cdRack(1, 3) = "Tattoo You"

The first item of the first dimension is an array for Beatles CDs while the second item of the first dimension is an array for Rolling Stones CDs. You could also add a third dimension if you wanted. Keeping with our CD rack analogy, you could picture this third dimension as a hallway with several rooms. Inside of each room would be a row of CDs racks. If you wanted your hallways to have 10 rooms, each with CD racks like the ones in the above example, you could declare your array as follows:
Dim cdRackHallway(0 to 9, 0 to 1, 0 to 3) As String

In Visual Basic 6.0, you can create arrays with up to 60 dimensions. In Visual Basic .NET, the maximum number of dimensions an array can have is 32. Most arrays you will need to deal with will only be one or two dimensions. Multidimensional arrays can require a decent amount of memory, so use them with care, especially large multidimensional arrays.

Lastly, for multidimensional arrays it should be noted that only the last dimension can be resized. That means that given our example above, once we created the array with two CD racks, we would not be able to add more racks, we would only be able to change the number of CDs each rack held. Example:

' Here we will define an array where the first dimension contains 2 elements and
two second dimension contains 4 elements
ReDim cdRack(0 to 1, 0 to 3) As String

' A CD rack for the Beatles
cdRack(0, 0) = "Rubber Soul"
cdRack(0, 1) = "Revolver"
cdRack(0, 2) = "The White Album"
cdRack(0, 3) = "Let It Be"

' A CD rack for the Rolling Stones
cdRack(1, 0) = "Sticky Fingers"
cdRack(1, 1) = "Beggars Banquet"
cdRack(1, 2) = "Let It Bleed"
cdRack(1, 3) = "Tattoo You"

ReDim Preserve cdRack(0 to 1, 0 to 4) As String

' Lets add another Beatles CD
cdRack(0, 4) = "Abby Road"

' Lets add another Rolling Stones CD
cdRack(1, 4) = "Exile on Main St."

Summary: In this unit we learned the looping structure available in Visual Basic. We described each looping structure with example, syntax, rules and behaviour. We also learned the differences between a fixed-size and dynamic array, how to properly declare each one, how to access them, how to loop through them, how to erase them, and a few other things.
3.1 Possibilities of Errors
Run-time errors are an expensive fact of life both during development and following the release of software to customers. Errors take time to find and time to fix. Users soon become frustrated when software fails to function as expected without adequate warning or explanation putting your reputation at stake. It is therefore extremely important that you consider carefully how your application will handle and report errors. Designing and implementing an effective error handling strategy can accelerate development and significantly reduce development and support costs with maintaining (or even enhancing) your reputation.

Unless we take deliberate steps in our code to trap and handle errors, Visual Basic (VB) will ‘handle’ them for us. It does so by displaying a simple message box prior to abruptly terminating your application without any ‘QueryUnload’, ‘Unload’ or ‘Terminate’ events!

To prevent our application users being unceremoniously dumped by VB’s default error handling behaviour, it is essential that all errors are correctly handled by your application. However, there is more to effective error handling than just placing an error trap in every procedure and displaying the property values of the VB ‘Err’ object whenever an error occurs.

Error Types
No matter how hard we try, errors do creep into our programs. These errors can be grouped into three categories:
1. Syntax errors
2. Run-time errors
3. Logic errors

- **Syntax errors** occur when you mistype a command or leave out an expected phrase or argument. Visual Basic detects these errors as they occur and even provides help in correcting them. You cannot run a Visual Basic program until all syntax errors have been corrected.

- **Run-time errors** are usually beyond your program’s control. Examples include: when a variable takes on an unexpected value (divide by zero), when a drive door is left open, or when a file is not found. Visual Basic allows you to trap such errors and make attempts to correct them.
- **Logic errors** are the most difficult to find. With logic errors, the program will usually run, but will produce incorrect or unexpected results. The Visual Basic debugger is an aid in detecting logic errors.

Implementing an effective VB error handling strategy ensures the following:
- Errors do not go undetected.
- Applications do not terminate abruptly without warning or explanation.
- Components are robust and predictable.
- After an error, application stability is restored.
- Error messages offer the user a meaningful description of an error.
- Error messages advise the user of the possible impact of an error.
- Error messages offer the user advice that may remedy a problem and avoid unnecessary support calls.
- Error messages advise the user on how to proceed.
- Error messages offer sufficient information for support personnel to quickly identify known problems.
- Legitimate run-time errors are not confused with software defects.
- Errors are logged to avoid incorrect reporting by users during support calls.
- Error logs can be viewed and cleared easily.
- The source code location of an error can quickly be determined.
- The cause of an error can be determined more easily allowing defects to be corrected more easily.
- The user/system is not swamped by repeated error reporting.

**Note:** For our VB error handling strategy to be successful, it should be implemented from the outset of a project. Developers must also understand the business reasons why error handling is important and necessary in addition to being thoroughly conversant with VB's error handling behavior and functionality.

### 3.2 Coding Conventions and Putting Comments

Some ways to minimize errors:
- Design our application carefully. More design time means less debugging time.
- Use comments where applicable to help you remember what you were trying to do.
- Use consistent and meaningful naming conventions for your variables, objects, and procedures.

#### Run-Time Error Trapping and Handling

- Run-time errors are trappable. That is, Visual Basic recognizes an error has occurred and enables you to trap it and take corrective action. If an error occurs and is not trapped, your program will usually end in a rather unceremonious manner.
- Error trapping is enabled with the On Error statement:
  ```vbscript
  On Error GoTo errlabel
  ```
  Yes, this uses the dreaded GoTo statement! Any time a run-time error occurs following this line, program control is transferred to the line labeled errlabel. Recall a labeled line is simply a line with the label followed by a colon (:)..
- The best way to explain how to use error trapping is to look at an outline of an example procedure with error trapping.

```vbscript
Sub SubExample()
  . [Declare variables, ...]
  .
End Sub
```
On Error GoTo HandleErrors
.
. [Procedure code]
.
Exit Sub
HandleErrors:
.
. [Error handling code]
.
End Sub

Once you have set up the variable declarations, constant definitions, and any other
procedure preliminaries, the On Error statement is executed to enable error trapping.
Your normal procedure code follows this statement. The error handling code goes at
the end of the procedure, following the HandleErrors statement label. This is the code
that is executed if an error is encountered anywhere in the Sub procedure. Note you
must exit (with Exit Sub) from the code before reaching the HandleErrors line to avoid
inadvertent execution of the error handling code.

Since the error handling code is in the same procedure where an error occurs, all
variables in that procedure are available for possible corrective action. If at some time
in your procedure, you want to turn off error trapping, that is done with the following
statement:
On Error GoTo 0

Once a run-time error occurs, we would like to know what the error is and attempt
to fix it. This is done
in the error handling code.

Visual Basic offers help in identifying run-time errors. The Err object returns, in its
Number property (Err.Number), the number associated with the current error
condition. (The Err function has other useful properties that we won’t cover here -
consult on-line help for further information.) The Error() function takes this error
number as its argument and returns a string description of the error. Consult online
help for Visual Basic run-time error numbers and their descriptions.

Once an error has been trapped and some action taken, control must be returned to
your application. That control is returned via the Resume statement. There are three
options:

Resume Lets you retry the operation that caused the error. That is, control is returned
to the line where the error occurred. This could be dangerous in that, if the error has
not been corrected (via code or by the user), an infinite loop between the error handler
and the procedure code may result.
Resume Next Program control is returned to the line immediately following the line
where the error occurred.
Resume label Program control is returned to the line labeled label.

Be careful with the Resume statement. When executing the error handling portion of
the code and the end of the procedure is encountered before a Resume, an error
occurs. Likewise, if a Resume is encountered outside of the error handling portion of
the code, an error occurs.

General Error Handling Procedure

Development of an adequate error handling procedure is application dependent.
You need to know what type of errors you are looking for and what corrective actions
must be taken if these errors are encountered. For example, if a 'divide by zero' is
found, you need to decide whether to skip the operation or do something to reset the
offending denominator.

What we develop here is a generic framework for an error handling procedure. It
simply informs the user that an error has occurred, provides a description of the error,
and allows the user to Abort, Retry, or Ignore. This framework is a good starting point for designing custom error handling for your applications.

The generic code (begins with label **HandleErrors**) is:

```
HandleErrors:
Select Case MsgBox(Error(Err.Number), vbCritical + vbAbortRetryIgnore, 
"Error Number" + Str(Err.Number))
Case vbAbort
Resume ExitLine
Case vbRetry
Resume
Case vbIgnore
Resume Next
End Select
ExitLine:
Exit Sub
```

Let's look at what goes on here. First, this routine is only executed when an error occurs. A message box is displayed, using the Visual Basic provided error description `[Error(Err.Number)]` as the message, uses a **critical icon** along with the **Abort**, **Retry**, and **Ignore** buttons, and uses the error number `[Err.Number]` as the title. This message box returns a response indicating which button was selected by the user. If Abort is selected, we simply exit the procedure. (This is done using a *Resume* to the line labeled *ExitLine*. Recall all error trapping must be terminated with a Resume statement of some kind.) If Retry is selected, the offending program line is retried (in a real application, you or the user would have to change something here to correct the condition causing the error). If Ignore is **Error-Handling, Debugging and File Input/Output** selected, program operation continues with the line following the error causing line.

- To use this generic code in an existing procedure, you need to do three things:

  1. Copy and paste the error handling code into the end of your procedure.
  2. Place an *Exit Sub* line immediately preceding the **HandleErrors** labeled line.
  3. Place the line, **On Error GoTo HandleErrors**, at the beginning of your procedure.

For example, if your procedure is the **SubExample** seen earlier, the modified code will look like this:

```
Sub SubExample()
[Declare variables, ...]
On Error GoTo HandleErrors
[Procedure code]
Exit Sub

HandleErrors:
Select Case MsgBox(Error(Err.Number), vbCritical + vbAbortRetryIgnore, 
"Error Number" + Str(Err.Number))
Case vbAbort
Resume ExitLine
Case vbRetry
Resume
Case vbIgnore
Resume Next
End Select
ExitLine:
Exit Sub
```

```
Again, this is a very basic error-handling routine. You must determine its utility in your applications and make any modifications necessary. Specifically, you need code to clear error conditions before using the Retry option.

- One last thing. Once you've written an error handling routine, you need to test it to make sure it works properly. But, creating run-time errors is sometimes difficult and perhaps dangerous. Visual Basic comes to the rescue! The Visual Basic Err object has a method (Raise) associated with it that simulates the occurrence of a run-time error. To cause an error with value Number, use:

Err.Raise Number

- We can use this function to completely test the operation of any error handler we write. Don’t forget to remove the Raise statement once testing is completed, though! And, to really get fancy, you can also use Raise to generate your own ‘application-defined’ errors. There are errors specific to your application that you want to trap.

- To clear an error condition (any error, not just ones generated with the Raise method), use the method Clear:

Err.Clear

**Simple Error Trapping Example**

1. Start a new project. Add a text box and a command button.

2. Set the properties of the form and each control:

   **Form1:**
   - BorderStyle 1-Fixed Single
   - Caption Error Generator
   - Name frmError

   **Command1:**
   - Caption Generate Error
   - Default True
   - Name cmdGenError

   **Text1:**
   - Name txtError
   - Text [Blank]

The form should look something like this:

3. Attach this code to the cmdGenError_Click event.

   Private Sub cmdGenError_Click()
   On Error GoTo HandleErrors
   Err.Raise Val(txtError.Text)
   Err.Clear
   Exit Sub
   HandleErrors:
   Select Case MsgBox(Err.Number), vbCritical +
vbAbortRetryIgnore, "Error Number" + Str(Err.Number))
Case vbAbort
Resume ExitLine
Case vbRetry
Resume
Case vbIgnore
Resume Next
End Select
ExitLine:
Exit Sub
End Sub

In this code, we simply generate an error using the number input in the text box. The generic error handler then displays a message box which you can respond to in one of three ways.

4. Save your application. Try it out using some of these typical error numbers (or use numbers found with on-line help). Notice how program control changes depending on which button is clicked.

3.3 Debugging, Handling Errors
We now consider the search for, and elimination of, logic errors. These are errors that don’t prevent an application from running, but cause incorrect or unexpected results. Visual Basic provides an excellent set of debugging tools to aid in this search.

- Debugging a code is an art, not a science. There are no prescribed processes that you can follow to eliminate all logic errors in your program. The usual approach is to eliminate them as they are discovered.

- What we’ll do here is present the debugging tools available in the Visual Basic environment (several of which appear as buttons on the toolbar) and describe their use with an example. You, as the program designer, should select the debugging approach and tools you feel most comfortable with.

- The interface between your application and the debugging tools is via three different debug windows: the Immediate Window, the Locals Window, and the Watch Window. These windows can be accessed from the View menu (the Immediate Window can be accessed by pressing Ctrl+G). Or, they can be selected from the Debug Toolbar (accessed using the Toolbars option under the View menu):

  ![Debug Toolbar](image)

- All debugging using the debug windows is done when your application is in break mode. You can enter break mode by setting breakpoints, pressing Ctrl+Break, or the program will go into break mode if it encounters an untrapped error or a Stop statement.

- Once in break mode, the debug windows and other tools can be used to:
  - Determine values of variables
  - Set breakpoints
  - Set watch variables and expressions
  - Manually control the application
- Determine which procedures have been called
- Change the values of variables and properties
  Immediate
  Locals
  Watch

**Debugging Example**
1. Unlike other examples, we’ll do this one as a group. It will be used to demonstrate use of the debugging tools.

2. The example simply has a form with a single command button. The button is used to execute some code. We won’t be real careful about proper naming conventions and such in this example.

3. The code attached to this button’s Click event is a simple loop that evaluates a function at several values.

```vba
Private Sub Command1_Click()
    Dim X As Integer, Y As Integer
    X = 0
    Do
        Y = Fcn(X)
        X = X + 1
    Loop While X <= 20
End Sub
```

This code begins with an X value of 0 and computes the Y value using the general integer function `Fcn`. It then increments X by 1 and repeats the Loop. It continues looping While X is less than or equal to 20. The function `Fcn` is computed using:

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

Admittedly, this code doesn’t do much, especially without any output, but it makes a good example for looking at debugger use. Set up the application and get ready to try debugging.

**Using the Debugging Tools**
- There are several debugging tools available for use in Visual Basic. Access to these tools is provided with both menu options and buttons on the Debug toolbar. These tools include breakpoints, watch points, calls, step into, step over, and step out.

- The simplest tool is the use of direct prints to the immediate window.
- Printing to the Immediate Window:
You can print directly to the **immediate window** while an application is running. Sometimes, this is all the debugging you may need. A few carefully placed print statements can sometimes clear up all logic errors, especially in small applications.

To print to the immediate window, use the **Print** method:

```
Debug.Print [List of variables separated by commas or semi-colons]
```

Debug.Print Example:

1. Place the following statement in the **Command1_Click** procedure after the line calling the general procedure **Fcn**:

   ```vba
   Debug.Print X; Y
   ```

   and run the application.

2. Examine the immediate window. Note how, at each iteration of the loop, the program prints the value of X and Y. You could use this information to make sure X is incrementing correctly and that Y values look acceptable.

3. Remove the Debug.Print statement.

   - **Breakpoints**:

   ```vba
   Breakpoint
   ```

   debug window. In many applications, we want to stop the application while it is running, examine variables and then continue running. This can be done with **breakpoints**.

   A breakpoint is a line in the code where you want to stop (temporarily) the execution of the program, that is force the program into break mode. To set a breakpoint, put the cursor in the line of code you want to break on. Then, press `<F9>` or click the **Breakpoint** button on the toolbar or select **Toggle Breakpoint** from the **Debug** menu. The line will be highlighted.

   When you run your program, Visual Basic will stop when it reaches lines with breakpoints and allow you to use the immediate window to check variables and expressions. To continue program operation after a breakpoint, press `<F5>`, click the **Run** button on the toolbar, or choose **Start** from the **Run** menu.

   You can also change variable values using the immediate window. Simply type a valid Basic expression. This can sometimes be dangerous, though, as it may change program operation completely.

**Breakpoint Example:**

1. Set a breakpoint on the **X = X + 1** line in the sample program. Run the program.

2. When the program stops, display the immediate window and type the following line:

   ```vba
   Print X;Y
   ```

3. The values of these two variables will appear in the debug window. You can use a question mark (?) as shorthand for the command **Print**, if you’d like. Restart the application. Print the new variable values.
4. Try other breakpoints if you have time. Once done, all breakpoints can be cleared by Ctrl+Shift+F9 or by choosing Clear All Breakpoints from the Debug menu. Individual breakpoints can be toggled using <F9> or the Breakpoint button on the toolbar.

- Viewing Variables in the Locals Window:

The locals window shows the value of any variables within the scope of the current procedure. As execution switches from procedure to procedure, the contents of this window changes to reflect only the variables applicable to the current procedure. Repeat the above example and notice the values of X and Y also appear in the locals window.

- Watch Expressions:

The Add Watch option on the Debug menu allows you to establish watch expressions for your application. Watch expressions can be variable values or logical expressions you want to view or test. Values of watch expressions are displayed in the watch window.

In break mode, you can use the Quick Watch button on the toolbar to add watch expressions you need. Simply put the cursor on the variable or expression you want to add to the watch list and click the Quick Watch button. Watch expressions can be edited using the Edit Watch option on the Debug menu.

Watch Expression Example:

1. Set a breakpoint at the \( X = X + 1 \) line in the example.

2. Set a watch expression for the variable X. Run the application. Notice X appears in the watch window. Every time you re-start the application, the value of X changes.

3. At some point in the debug procedure, add a quick watch on Y. Notice it is now in the watch window.

4. Clear the breakpoint. Add a watch on the expression: \( X = Y \). Set Watch Type to "Break When Value Is True." Run the application. Notice it goes into break mode and displays the watch window whenever \( X = Y \). Delete this last watch expression.

- Call Stack:

Selecting the Call Stack button from the toolbar (or pressing Ctrl+L or selecting Call Stack from the View menu) will display all active procedures, that is those that have
not been exited. **Call Stack** helps you unravel situations with nested procedure calls to give you some idea of where you are in the application.

- **Call Stack Example:**
  1. Set a breakpoint on the `Fcn = Cint()` line in the general function procedure. Run the application. It will break at this line.
  2. Press the **Call Stack** button. It will indicate you are currently in the `Fcn` procedure which was called from the `Command1_Click` procedure. Clear the breakpoint.

- **Single Stepping (Step Into):**

While at a breakpoint, you may execute your program one line at a time by pressing `<F8>`, choosing the **Step Into** option in the **Debug** menu, or by clicking the **Step Into** button on the toolbar. This process is **single stepping**. It allows you to watch how variables change (in the locals window) or how your form changes, one step at a time. You may step through several lines at a time by using **Run To Cursor** option. With this option, click on a line below your current point of execution. Then press **Ctrl+<F8>** (or choose **Run To Cursor** in the **Debug** menu). the program will run through every line up to the cursor location, then stop.

- **Step Into Example:**
  1. Set a breakpoint on the `Do` line in the example. Run the application.
  2. When the program breaks, use the **Step Into** button to single step through the program.
  3. At some point, put the cursor on the `Loop While` line. Try the **Run To Cursor** option (press **Ctrl+<F8>**).

- **Procedure Stepping (Step Over):**

While single stepping your program, if you come to a procedure call you know functions properly, you can perform **procedure stepping**. This simply executes the entire procedure at once, rather than one step at a time. To move through a procedure in this manner, press **Shift+<F8>**, choose **Step Over** from the **Debug** menu, or press the **Step Over** button on the toolbar.

- **Step Over Example:**
  1. Run the previous example. Single step through it a couple of times.
  2. One time through, when you are at the line calling the `Fcn` function, press the **Step Over** button. Notice how the program did not single step through the function as it did previously.

- **Function Exit (Step Out):**

While stepping through your program, if you wish to complete the execution of a **function** you are in, without stepping through it line-by-line, choose the **Step Out**
option. The function will be completed and you will be returned to the procedure accessing that function. To perform this step out, press Ctrl+Shift+F8, choose Step Out from the Debug menu, or press the Step Out button on the toolbar. Try this on the previous example.

**Debugging Strategies**

- We’ve looked at each debugging tool briefly. Be aware this is a cursory introduction. Use the on-line help to delve into the details of each tool described. Only through lots of use and practice can you become a proficient debugger.

There are some guidelines to doing a good job, though.

- My first suggestion is: keep it **simple**. Many times, you only have one or two bad lines of code. And you, knowing your code best, can usually quickly narrow down the areas with bad lines. Don’t set up some elaborate debugging procedure if you haven’t tried a simple approach to find your error(s) first. Many times, just a few intelligently-placed Debug.Print statements or a few examinations of the immediate and locals windows can solve your problem.

- A tried and true approach to debugging can be called **Divide and Conquer**. If you’re not sure where your error is, guess somewhere in the middle of your application code. Set a breakpoint there. If the error hasn’t shown up by then, you know it’s in the second half of your code. If it has shown up, it’s in the first half. Repeat this division process until you’ve narrowed your search.

- And, of course, the best debugging strategy is to be careful when you first design and write your application to minimize searching for errors later.

### 3.4 Exit Statements

Exits a procedure or block and transfers control immediately to the statement following the procedure call or the block definition.

Exit { Do | For | Function | property | Select | Sub | Try | While }

**Do**

Immediately exits the **Do** loop in which it appears. Execution continues with the statement following the **Loop** statement. **Exit Do** can be used only inside a **Do** loop. When used within nested **Do** loops, **Exit Do** exits the innermost loop and transfers control to the next higher level of nesting.

**For**

Immediately exits the **For** loop in which it appears. Execution continues with the statement following the **Next** statement. **Exit For** can be used only inside a **For...Next** or **For Each...Next** loop. When used within nested **For** loops, **Exit For** exits the innermost loop and transfers control to the next higher level of nesting.
Function
Immediately exits the Function procedure in which it appears. Execution continues with the statement following the statement that called the Function procedure. Exit Function can be used only inside a Function procedure.

Property
Immediately exits the Property procedure in which it appears. Execution continues with the statement that called the Property procedure, that is, with the statement requesting or setting the property's value. Exit Property can be used only inside a property's Get or Set procedure.

Select
Immediately exits the Select Case block in which it appears. Execution continues with the statement following the End Select statement. Exit Select can be used only inside a Select Case statement.

Sub
Immediately exits the Sub procedure in which it appears. Execution continues with the statement following the statement that called the Sub procedure. Exit Sub can be used only inside a Sub procedure.

Try
Immediately exits the Try or Catch block in which it appears. Execution continues with the Finally block if there is one, or with the statement following the End Try statement otherwise. Exit Try can be used only inside a Try or Catch block, and not inside a Finally block.

While
Immediately exits the While loop in which it appears. Execution continues with the statement following the End While statement. Exit While can be used only inside a While loop. When used within nested While loops, Exit While transfers control to the loop that is one nested level above the loop where Exit While occurs.

Do not confuse Exit statements with End statements. Exit does not define the end of a statement.

Example
The following example uses the Exit statement to exit a For...Next loop, a Do loop, and a Sub procedure.

Sub exitStatementDemo()
    Dim demoNum As Single
    ' Set up an infinite loop.
    Do
        For i As Integer = 1 To 1000000
    Loop
demoNum = Int(Rnd() * 100)
Select Case demoNum
    Case 7 : Exit For
    Case 29 : Exit Do
    Case 54 : Exit Sub
End Select
Next i
Loop
End Sub

3.5 List of Some Trappable Errors

Return without GoSub (Error 3)
A Return statement does not have a corresponding GoSub statement. Unlike For...Next, While...Wend, and Sub...End Sub, which are matched at compile time, GoSub and Return are matched at run time.

Invalid procedure call (Error 5)
An argument probably exceeds the range of permitted values. For example, the Sin function can only accept values within a certain range. Positive arguments less than 2147483648 are accepted, while 2147483648 generate this error.
This error may also occur if an attempt is made to call a procedure that is not valid on the current platform. For example, some procedures may only be valid for the Macintosh, or for Microsoft Windows, and so forth.

Overflow (Error 6)
Possible causes for this error are:
  • The result of an assignment, calculation, or data type conversion is too large to be represented within the range allowed for that type of variable.

  -or-

  • An assignment to a property exceeds the maximum value the property can accept.

Out of memory (Error 7)
More memory was required than is available or a 64K segment boundary was encountered. To prevent this error, try the following:
  • Close any unnecessary applications, documents, or source files that are in memory.
  • If you have extremely large modules or procedures, consider breaking them into smaller ones. This procedure doesn't save memory, but it can prevent hitting 64K segment boundaries.
  • If you are running Microsoft Windows in standard mode on an 80386 or 80486 computer, try running it in enhanced mode.
  • If you are running Microsoft Windows in enhanced mode, free up some disk space, or at least ensure that some space is available.
  • Eliminate terminate-and-stay-resident (tsr) programs.
  • Eliminate unnecessary device drivers.
  • Reduce the number of global variables.

Subscript out of range (Error 9)
You have referenced a nonexistent array element or collection member. The subscript may be larger or smaller than the range of possible subscripts, or the array may not have dimensions assigned at this point in the application.

Duplicate definition (Error 10)
This error usually occurs only when generated from code, as in the following example:
Error 10
However the error may also occur if you try to use ReDim to change the number of elements of a fixed-size array. For example, in the following code, the fixed array FixedArr is received by SomeArr in the procedure NextOne, then an attempt is made to resize SomeArr:

```vbnet
Sub FirstOne
    Static FixedArr(25) As Integer  ' Create a fixed-size array
    NextOne FixedArr()             ' and pass it to another sub.
End Sub

Sub NextOne(SomeArr() As Integer)
    ReDim SomeArr(35)              ' Duplicate definition occurs here.
    . . .
End Sub
```

Division by zero (Error 11)
The value of an expression being used as a divisor is zero. Check the spelling of variables in the expression. A misspelled variable name can implicitly create a numeric variable that is initialized to zero. Check previous operations on variables in the expression, especially those passed into the procedure as arguments from other procedures.

Type mismatch (Error 13)
Possible causes for this error are:

- The variable or property is not of the correct type. For example, a variable that requires an integer value can't accept a string value.
- An object has been passed to a procedure that is expecting a single property or value.
- A module or project name was used where an expression was expected, for example:
  ```vbnet
  Debug.Print MyModule
  ```
- You attempted to mix traditional Basic error handling with variant values having the Error subtype (10, vbError); for example: Error CVErr(n)
- A CVErr value can't be converted to Date or numeric types (Integer, Long, and so on); for example:
  ```vbnet
  MyVar=CInt(CVErr(9))
  ```
  At run time, this error typically indicates that a Variant used in an expression has an incorrect subtype, or that a Variant containing an array appeared in a Print statement.

Out of string space (Error 14)
Your system may have run out of memory, which has prevented a string from being allocated. Similarly, expressions requiring that temporary strings be created for evaluation may be causing this error. For example, the following code will cause an Out of string space error:

```vbnet
MyString = "Hello"
For Count = 1 To 100
    MyString = MyString & MyString
Next Count
```

Visual Basic lets you allocate a string with a maximum length of 65,535 characters. However, in executing statements at run time, the host application needs to use some
string space for overhead. The amount of overhead varies among hosts, but should not exceed 50 bytes. If you need to allocate a string of the maximum length your host can support, reduce the string size by 50 bytes, then increase the length incrementally until this error is generated again. The value immediately preceding the error represents the host's maximum string length.

```vba
Dim MyString As String * 65485
    ' Start with (65535-50).
    ' On successive runs, increment
    ' length until "Out of string
    ' space" error occurs.
Sub MySub
    MyString = "string" ' Error occurs here when the
End Sub                  ' maximum length is exceeded.
```

**String formula too complex (Error 16)**
A string expression is too complicated. Strings not assigned to variables (such as those returned by functions) are assigned to temporary locations during string expression evaluation. Having a large number of these strings can cause this error. Try assigning these strings to variables and use the variables in the expression instead.

**Can't perform requested operation (Error 17)**
The requested operation can't be performed because it would invalidate the current state of the project. For example, the error occurs if you use the References dialog box (on the Tools menu, click References) to add a reference to a new project or object library while a program is in break mode.

**User interrupt occurred (Error 18)**
A CTRL+BREAK or other interrupt key has been pressed by the user.

**Resume without error (Error 20)**
A Resume statement has been encountered, but it is either outside the error handler code, or it was encountered while there was no error-handling routine active.

**Out of stack space (Error 28)**
Possible causes for this error are:
- Too many active Function or Sub calls. Check that general recursive procedures are not nested too deeply and that they terminate properly.
- Local variables require more local variable space than is available. Try declaring some variables at the module level instead. You can also declare all variables in the procedure static by preceding the Property, Sub, or Function keyword with Static. Or, you can use the Static statement to declare individual static variables within procedures.
- Fixed-length strings use more stack space than variable-length strings. Try redefining some of your fixed-length strings as variable-length strings.
- Too many nested DoEvents statements.

Use the Calls dialog box to view which procedures are active (on the stack). To display the Calls dialog box, select the button to the right of the Procedures box in the Debug window.
Sub or function not defined (Error 35)
A Sub, Function, or Property procedure is called but is not defined. Possible causes for this error are:

- You have misspelled the name of your procedure.
- The specified procedure is not visible to the calling procedure. Procedures declared Private in one module can't be called from procedures outside the module. If Option Private Module is in effect, procedures in the module are not available to other projects. Choose Find from the Edit menu to locate the procedure.
- You have declared a dynamic-link library (DLL) routine, but the routine is not in the specified library.

Error in loading DLL (Error 48)
The specified dynamic-link library (DLL) can't be loaded. This is usually because the file specified with the Lib clause in the Declare statement is not a valid DLL.

Possible causes for this error are:

- The file is not DLL-executable.
- The file is not a Microsoft Windows DLL.
- The file is an old Microsoft Windows DLL that is incompatible with Microsoft Windows protect mode.
- The DLL references another DLL that is not present.
- The DLL or one of the referenced DLLs is not in a directory specified by your path.

Bad DLL calling convention (Error 49)
Your program is calling a routine in a dynamic-link library (DLL) that either is being passed the wrong type or number of arguments or does not use the Pascal calling convention. Make sure that the arguments passed to the DLL routine exactly match the arguments expected by the routine. If the DLL routine expects arguments by value, then make sure ByVal is specified for those arguments in the declaration for the routine.

Internal error (Error 51)
An internal malfunction has occurred in Visual Basic. Unless this call was generated by the Error statement, contact Microsoft Product Support Services to report the conditions under which the message appeared.

Bad file name or number (Error 52)
A statement refers to a file with a file number or file name that is:

- An invalid name or number
- Not specified in the Open statement
- Specified in an Open statement, but has since been closed
- Out of the range of file numbers (1-511)

- The name should not contain any spaces. The following names are reserved and can't be used for files or directories: CON, AUX, COM1, COM2, COM3, COM4, LPT1, LPT2, LPT3, PRN, and NUL. For instance, if you try to name a file PRN in an Open statement, the default printer will simply become the destination for Print #, and Write # statements directed to the file number specified in the Open statement.
- The following are examples of valid Microsoft Windows file names:
On the Macintosh, a file can have any character except the colon (\:), and may contain spaces. Null characters [Chr(0)] are not allowed in any file names.

**File not found (Error 53)**
Possible causes for this error at run time are:
- A statement (for example, Kill, Name, or Open) refers to a file that does not exist.
- An attempt has been made to call a procedure in a dynamic-link library (DLL), but the library file name specified in the Lib clause of the Declare statement can't be found.

In the development environment, this error occurs if you attempt to open a project or load a text file that does not exist.

**Bad file mode (Error 54)**
Possible causes for this error are:
- A Put or Get statement is specifying a sequential file. Note that Put and Get can only refer to files opened for Random access.
- A Print # statement specifies a file opened for an access mode other than Output or Append.
- An Input # statement specifies a file opened for an access mode other than Input.
- Any attempt to write to a read-only file.

**File already open (Error 55)**
Possible causes for this error are:
- A sequential-output mode Open statement was executed for a file that is already open.
- A statement (for example Kill, SetAttr, or Name) refers to an open file.

**Device I/O error (Error 57)**
An input or output error occurred while your program was using a device such as a printer or disk drive.

**File already exists (Error 58)**
At run time, this error occurs when the new file name (for example, one specified in a Name statement) is identical to a file name that already exists. It also occurs when you use Save As to save a currently loaded project if the project name already exists.

**Bad record length (Error 59)**
The length of a record variable for a Get or Put statement does not match the length specified in the corresponding Open statement. Because a two-byte descriptor is always added to a variable-length string Put to a random access file, the variable-length string must be at least two characters shorter than the record length specified in the Len clause of the Open statement. Variant data types also require a two-byte descriptor. Variants containing variable-length strings require a four-byte descriptor. Therefore, for variable-length strings in a Variant, the string must be at least 4 bytes shorter than the record length specified in the Len clause.
Disk full (Error 61)
Possible causes for this error are:

- There is not enough room on the disk for the completion of a Print #, Write #, or Close operation.
- There is not enough room on the disk to create required files.

To work around this situation, move some files to another disk, or delete some files.

Input past end of line (Error 62)
An Input # or Line Input # statement is reading from a file in which all data has already been read or from an empty file. To avoid this error, use the EOF function (immediately before the Input # statement) to detect the end of file.

Bad record number (Error 63)
The record number in a Put or Get statement is less than or equal to zero.

Too many files (Error 67)
Possible causes for this error are:

- There is a limit to the number of disk files that can be open at one time. For Microsoft Windows, this limit is a function of the Files= setting in your CONFIG.SYS file. Increase that number and reboot.
- The operating system has a limit to the number of files in the root directory (usually 512). If your program is opening, closing, or saving files in the root directory, change your program so that it uses a subdirectory.
- On the Macintosh, the standard limit is 40 files. This limit can be changed using a utility to modify the MaxFiles parameter of the boot block.

Device unavailable (Error 68)
The device you are trying to access is either not online or does not exist.

Permission denied (Error 70)
An attempt was made to write to a write-protected disk or to access a locked file. For example, this error will occur if an Open For Output statement is performed on a write-protected file.

Disk not ready (Error 71)
There is either no disk in the drive specified or the drive door is open. Insert a disk in the drive, close the door, and retry the operation.

Can't rename with different drive (Error 74)
You can't use the Name statement to rename a file with a new drive designation. Use FileCopy to write the file to another drive, and delete the old file with a Kill statement.

Path/File access error (Error 75)
During a file- or disk-access operation (for example, Open, MkDir, ChDir, or RmDir), the operating system could not make a connection between the path and the file name.
Make sure the file specification is formatted correctly. A file name can contain a fully-qualified or relative path. A fully-qualified path starts with the drive name (if the path is on another drive) and lists the explicit path from the root to the file. Any path that is not fully qualified is relative to the current drive and directory. This error can also occur while attempting to save a file that would replace an existing read-only file.

Path not found (Error 76)
During a file- or disk-access operation (for example, Open, MkDir, ChDir, or RmDir), the operating system was unable to find the specified path. The error also occurs in the debugging environment if you attempt to open a project or insert a text file with an invalid path. Make sure the path is typed correctly.
Object variable not set (Error 91)
You are attempting to use an object variable that is not yet referencing a valid object, or one that has been set to Nothing. Specify or respecify a reference for the object variable. For example, if the Set statement were omitted in the following code, an error would be generated:

Dim MyObject As Object ' Creates object variable.
    Set MyObject = Sheets(1) ' Creates valid object reference.

For Loop not initialized (Error 92)
You've jumped into the middle of a For...Next loop. Placing labels inside a For...Next loop is not recommended.

Invalid pattern string (Error 93)
The pattern string specified in the Like operation of a search is invalid. A common example of an invalid character list expression is [a-b], where the right bracket is missing.

Invalid use of Null (Error 94)
You are trying to obtain the value of a variant variable or an expression that is Null. Null is a variant subtype used to indicate that a data item contains no valid data. For example:

    MyVar = Null
    For Count = 1 To MyVar
        ... 
    Next Count

Can't load module; invalid format (Error 323)
The module you attempted to load is not a text module. Some versions of Visual Basic permit you to save code in both binary and text formats. If possible, reload the file in the application in which it was last saved and save it as text. This error code applies to Microsoft Excel for Windows 95, version 7.0 only.

Property or method not found (Error 423)
Object.method or object.property is referred to, but method or property is not defined, or you may have misspelled the name of the object. To see what properties and methods are defined for an object, choose the Object Browser from the View menu. Select the appropriate library and object to display available properties and methods.

Object required (Error 424)
You have referred to an object property or method, but have not provided a valid object qualifier.

Class doesn't support OLE Automation (Error 430)
The object specified in the GetObject or CreateObject function call was found, but has not exposed a programmability interface. Therefore you can't write code to control this object's behavior. Check the documentation of the application that created the object for limitations on the use of OLE Automation with this class of object.
Object doesn't support this property or method (Error 438)
This method or property does not exist for this OLE automation object. See the object's documentation for more information on the object and to check the spellings of properties and methods.

OLE Automation error (Error 440)
An error occurred while executing a method or accessing a property of an object variable. The error was reported by the application that created the object.

Object doesn't support this action (Error 445)
This method or property is not supported by this object. See the object's documentation for more information on the object and to check the spellings of properties and methods.

Object doesn't support named arguments (Error 446)
Arguments can only be specified by position when performing methods on this object. See the object's documentation for more information on argument positions and types.

Object doesn't support current locale settings (Error 447)
The object you are attempting to access does not support the locale setting for the current project. For example, if your current project has the locale setting Canadian French, the object you are trying to access must support that locale setting. Check which locale settings the object supports. Also note that the object may rely on national language support in a dynamic-link library (DLL), for example OLE2NLS.DLL. If so, you may need a more recent version that supports the current project locale.

Named argument not found (Error 448)
You specified a named argument, but the procedure was not defined to accept an argument by that name. Check the spelling of the argument name.

Argument not optional (Error 449)
The number and types of arguments must match those expected. For instance, the Left function requires two arguments, the first representing the character string being operated on, and the second representing the number of characters to return from the left side of the string. Because neither argument is optional, both must be supplied. An argument can only be omitted from a call to a user-defined procedure if it was declared Optional in the procedure declaration.

Wrong number of arguments (Error 450)
The number of arguments in the call to the procedure was not the same as the number of arguments expected by the procedure. Check the argument list in the call against the procedure declaration.

Object not a collection (Error 451)
You have specified an operation or property that is exclusive to collections, but the object is not a collection. Check the spelling of the object or property name, or verify that the object is a collection.

Invalid ordinal (Error 452)
Your call to a dynamic-link library (DLL) indicated to use a number instead of a procedure name, using the #num syntax. However, an attempt to convert the expression num to an ordinal failed, or the num specified does not specify any function.
in the DLL. Check to make sure the expression represents a valid number, or call the procedure by name.

**Specified DLL function not found (Error 453)**
The dynamic-link library (DLL) in a user library reference was found, but the DLL function specified was not found within the DLL. An invalid ordinal may have been specified in the function declaration. Also, the DLL may have the right name but is not the version that contains the specified function.

**Code resource not found (Error 454)**
A call was made to a procedure in a code resource, but the code resource could not be found. This error can only occur on the Macintosh operating system.

**Code resource lock error (Error 455)**
A call was made to a procedure in a code resource. The code resource was found, but an error occurred when an attempt was made to lock the resource. Check for an error returned by HLock (for example, "Illegal on empty handle" or "Illegal on free block"). This error can only occur on the Macintosh operating system.

**[Object] does not have [property name] property (Error 1000)**
The property does not exist for this object. To see a list of properties for this object, choose Object Browser from the View menu, and click the question mark button in the Object Browser dialog box to display the Visual Basic Help topic for this object.

**[Object] does not have [method name] method (Error 1001)**
The method does not exist for this object. To see a list of methods for this object, choose Object Browser from the View menu, and click the question mark button in the Object Browser dialog box to display the Visual Basic Help topic for this object.

**Missing required argument [argument] (Error 1002)**
The method expected a required argument that does not exist. Add the argument to the code. To see a list of required arguments, choose Object Browser from the View menu, and click the question mark button in the Object Browser dialog box to display the Visual Basic Help topic for this method.

**Invalid number of arguments (Error 1003)**
The method has the wrong number of arguments. This usually occurs when you use comma-separated position arguments (instead of named arguments), and you have too many arguments. To see the valid arguments for this method, choose Object Browser from the View menu, and click the question mark button in the Object Browser dialog box to display the Visual Basic Help topic for this method.

**[Method name] method of [object] class failed (Error 1004)**
An external error occurred, such as a failure to read or write from a file. The method cannot be used on the object. Possible reasons include the following:
- An argument contains a value that isn't valid. A common cause of this problem is an attempt to access an object that doesn't exist [for example, you tried to use Workbooks(5) when there were only three workbooks open].
- The method cannot be used in the applied context. For example, some Range object methods require that the range contain data; if the range doesn't contain data, the method fails.
- An external error occurred, such as a failure to read or write from a file.
For more information about the method, search Help for the method name.
Unable to set the [property name] property of the [object] class (Error 1005)
The property cannot be changed. Possible reasons include the following:
- The value you're using for the property isn't valid (for example, you set a
  property to a string value, but the property requires a Boolean value).
- The property is read-only and can not be written to.

Unable to get the [property name] property of the [object] class (Error 1006)
The property cannot be changed. Possible reasons include:
- The value you are using for the property is not valid; for example, setting a
  property to a string value when the property requires a Boolean value.
- The property cannot be used in the applied context. For example, the code
  ActiveChart.Legend.Font.Color = RGB(255, 0, 0) will cause this error if the
  active chart does not contain a legend.

Mostly used error number and error description

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Overflow</td>
</tr>
<tr>
<td>9</td>
<td>Subscript out of range</td>
</tr>
<tr>
<td>11</td>
<td>Division by zero</td>
</tr>
<tr>
<td>13</td>
<td>Type mismatch</td>
</tr>
<tr>
<td>16</td>
<td>Expression too complex</td>
</tr>
<tr>
<td>20</td>
<td>Resume without error</td>
</tr>
<tr>
<td>52</td>
<td>Bad file name or number</td>
</tr>
<tr>
<td>53</td>
<td>File not found</td>
</tr>
<tr>
<td>55</td>
<td>File already open</td>
</tr>
<tr>
<td>61</td>
<td>Disk full</td>
</tr>
<tr>
<td>70</td>
<td>Permission denied</td>
</tr>
<tr>
<td>92</td>
<td>For loop not initialized</td>
</tr>
</tbody>
</table>

3.6 Summary:
In this unit we discussed about errors occurs in visual basic, and how we trap the
errors.
There are some guidelines to doing a good job
- keep it simple. Many times, we only have one or two bad lines of code. And
  we, knowing our code best, can usually quickly narrow down the areas with
  bad lines. Don’t set up some elaborate debugging procedure if you haven’t
  tried a simple approach to find your error(s) first. Many times, just a few
  intelligently-placed Debug.Print statements or a few examinations of the
  immediate and locals windows can solve your problem.
- A tried and true approach to debugging can be called Divide and Conquer. If
  we’re not sure where our error is, guess somewhere in the middle of your
  application code. Set a breakpoint there. If the error hasn’t shown up by then,
  we know it’s in the second half of your code. If it has shown up, it’s in the first
  half. Repeat this division process until we’ve narrowed our search.
- And, of course, the best debugging strategy is to be careful when we first
  design and write our application to minimize searching for errors later.
Unit 4: Sorting Techniques

Unit Contains

4.1 Introduction
4.2 Classification
4.3 Sorting Techniques
4.4 Implementing Algorithms
   - Bubblesort
   - Selectionsort
   - Quicksort
4.5 Summary

4.1 Introduction
Sorting using trees is generally pretty fast, but there are many other sorting algorithms that are less complicated and some of them are much faster. There are several reasons why sorting algorithms are so popular. First, sorting is a common task in computer applications. Almost any data is more meaningful if you can sort it and display it in various ways. Second, sorting algorithms demonstrate many important algorithmic techniques such as binary subdivision, recursion, and linear interpolation. Finally, different sorting algorithms behave differently for different data so no algorithm is best under all circumstances.

In computer science and mathematics, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

1. The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);
2. The output is a permutation, or reordering, of the input.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956. Although many consider it a solved problem, useful new sorting algorithms are still being invented (for example, library sort was first published in 2004). Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and lower bounds.

4.2 Classification
Sorting algorithms used in computer science are often classified by:

- Computational complexity (worst, average and best behaviour) of element comparisons in terms of the size of the list. For typical sorting algorithms good behavior is and bad behavior is. Ideal behavior for a sort is Comparison sorts, sort algorithms which only access the list via an abstract key comparison operation, always need comparisons in the worst case.
- Computational complexity of swaps (for "in place" algorithms).
Memory usage (and use of other computer resources). In particular, some sorting algorithms are "in place". This means that they need only or memory beyond the items being sorted and they don't need to create auxiliary locations for data to be temporarily stored, as in other sorting algorithms.

Recursion. Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).

Stability: stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).

Whether or not they are a comparison sort. A comparison sort examines the data only by comparing two elements with a comparison operator.

General method: insertion, exchange, selection, merging, etc. Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heapsort.

Adaptability: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

### 4.3 Sorting Techniques

Some short descriptions on each of the algorithms:

**Bubble Sort**
Exchange two adjacent elements if they are out of order. Repeat until array is sorted. This is a slow algorithm.

**Selection Sort**
Find the largest element in the array, and put it in the proper place. Repeat until array is sorted. This is also slow.

**Insertion Sort**
Scan successive elements for out of order item, then insert the item in the proper place. Sort small array fast, big array very slowly.

**Quicksort**
Partition array into two segments. The first segment all elements are less than or equal to the pivot value. The second segment all elements are greater or equal to the pivot value. Sort the two segments recursively. Quicksort is fastest on average, but sometimes unbalanced partitions can lead to very slow sorting.

**Mergesort**
Start from two sorted runs of length 1, merge into a single run of twice the length. Repeat until a single sorted run is left. Mergesort needs N/2 extra buffer. Performance is second place on average, with quite good speed on nearly sorted array. Mergesort is stable in that two elements that are equally ranked in the array will not have their relative positions flipped.

**Heapsort**
Form a tree with parent of the tree being larger than its children. Remove the parent from the tree successively. On average, Heapsort is third place in speed. Heapsort does not need extra buffer, and performance is not sensitive to initial distributions.

**Shellsort**
Sort every Nth element in an array using insertion sort. Repeat using smaller N values, until N = 1. On average, Shellsort is fourth place in speed. Shellsort may sort some distributions slowly.

### 4.4 Implementing Algorithms

**Bubblesort**

Bubblesort is a specialized algorithm designed for sorting items that are already mostly in sorted order. If only one or two items in your list are out of order, bubblesort is very fast. If the items in your list are initially arranged randomly, bubblesort is extremely slow. For this reason you should be careful when you use bubblesort.
The idea behind the algorithm is to scan through the list looking for two adjacent items that are out of order. When you find two such items, you swap them and continue down the list. You repeat this process until all of the items are in order.

Figure 1 shows a small list where the item 1 is out of order. During the first pass through the list, the algorithm will find that items 4 and 1 are out of order so it swaps them. During the next pass it finds that items 3 and 1 are out of order so it swaps them. During the third pass it swaps items 2 and 1 and the list is in its final order.

The way in which the item 1 seems to bubble towards the top of the list is what gives bubblesort its name.

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

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

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

**VB code for the improved bubblesort algorithm**

```vba
' min and max are the minimum and maximum indexes
' of the items that might still be out of order.
Sub BubbleSort(List() As Long, ByVal min As Integer, ByVal max As Integer)
    Dim last_swap As Integer
    Dim i As Integer
    Dim j As Integer
    Dim tmp As Long
    ' Repeat until we are done.
    Do While min < max
        ' Bubble up.
        last_swap = min - 1
        ' For i = min + 1 To max
        i = min + 1
        Do While i <= max
            ' Find a bubble.
```
If List(i - 1) > List(i) Then
    ' See where to drop the bubble.
    tmp = List(i - 1)
    j = i
    Do
        List(j - 1) = List(j)
        j = j + 1
        If j > max Then Exit Do
    Loop While List(j) < tmp
    List(j - 1) = tmp
    last_swap = j - 1
    i = j + 1
    Else
        i = i + 1
    End If
Loop
    ' Update max.
    max = last_swap - 1
    ' Bubble down.
    last_swap = max + 1
    ' For i = max - 1 To min Step -1
    i = max - 1
    Do While i >= min
        ' Find a bubble.
        If List(i + 1) < List(i) Then
            ' See where to drop the bubble.
            tmp = List(i + 1)
            j = i
            Do
                List(j + 1) = List(j)
                j = j - 1
                If j < min Then Exit Do
            Loop While List(j) > tmp
            List(j + 1) = tmp
            last_swap = j + 1
            i = j - 1
            Else
                i = i - 1
            End If
        Loop
        ' Update min.
        min = last_swap + 1
    Loop
End Sub

Selectionsort

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

The VB code for selectionsort is shown below-
Sub Selectionsort (List() As Long, min As Integer, max As Integer)
Dim i As Integer
Dim j As Integer
Dim best_value As Long
Dim best_j As Integer

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

While looking for the Ith smallest item, you must examine each of the N - I items that
you have not yet placed in their final positions. Then the total number of steps the
algorithm needs is:
N + (N - 1) + (N - 2) + ... + 1 = N * (N + 1) / 2
This function is on the order of N2. That means if you increase the number of items in
the list by a factor of 2, the run time of the algorithm will increase by a factor of roughly
22 = 4. There are several other sorting algorithms that require only about N * log(N)
steps (quicksort is one described below), so selectionsort is not a very fast algorithm
for large lists.
Selectionsort is fine for small lists, however. It is very simple so it is easy to program,
debug, and maintain over time. In fact it is so simple that it is actually faster than the
more complicated algorithms if the list you are sorting is very small. If your list contains
only a dozen or so items, selectionsort will probably be your best choice.

Quicksort

Quicksort is a recursive algorithm that uses a divide-and-conquer technique. While the
list of items to be sorted contains at least two items, quicksort divides it into two
sublists and recursively calls itself to sort the sublists.
The quicksort routine first checks to see if the list it is sorting contains fewer than two
items. If so, it simply returns.
Otherwise the subroutine picks an item from the list to use as a dividing point. It then
places all of the items that belong before this dividing point in the left part of the list. It
places all of the other items in right part of the list. The subroutine then recursively
calls itself to sort two smaller sublists.
There are several ways in which the quicksort routine might pick the dividing item. One
of the easiest is to simply use the first item in the sublist being sorted. If the list is
initially arranged randomly, that item will be a reasonable choice. Chances are good
that the item will belong somewhere in the middle of the list and the two sublists the
algorithm creates will be reasonably equal in size.
If the numbers are initially sorted or almost sorted, or if they are initially sorted in
reverse order, then this method fails miserably. In that case the first item in the list will
divide the list into one sublist that contains almost every item and another that will
contain almost no items. Since the larger sublist does not shrink much, the algorithm
makes little headway. In this case the algorithm will require on the order of N2 steps.
This is the same order of performance given by selectionsort, only this algorithm is much more complicated. A better method for selecting the dividing item is to choose one randomly. Then no matter how the items in the list are arranged, chances are the item you select will belong near the middle of the list and the sublists will be fairly evenly sized. As long as the sublists are fairly equal in size, the algorithm will require on the order of \( N \times \log(N) \) steps. It can be proven that this is the fastest time possible for a sorting algorithm that sorts using comparisons. By using a little randomness, this algorithm avoids the possibility of its worst case \( N^2 \) behavior and gives an expected case performance of \( N \times \log(N) \). Quicksort is very fast in practice as well as theory, so it is the favorite sorting algorithm of many programmers.

**VB code for the quicksort routine**

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

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

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

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

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

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

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

End Sub
List(lo) = med_value
Exit Do
End If

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

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

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

- If your list is more than 99% sorted already, use bubblesort.
- If you have a very small list (under 100 items or so), use selectionsort.
- Otherwise use quicksort.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubblesort</td>
<td>Very fast for lists that are almost sorted</td>
<td>Very slow sorting all other Lists</td>
</tr>
</tbody>
</table>
| Selectionsort | Very simple  
             | Easy to understand  
             | Very fast for small lists | Slow for large lists |
| Quicksort   | Very fast for large lists  
             | Trouble if there are lots of duplicate data values | |

4.5 Summary:
In summary we can say that
- Selectionsort is fastest for small lists (use many repetitions).
- Quicksort is faster for larger lists.
- Quicksort is faster than countingsort if the largest item is large (like 30,000) compared to the number of items (like 1,000).
- Bubblesort is fastest if the list starts mostly sorted.
References:

CS – 07 Programming in VB
Block 3: Object Oriented & Event Driven Programming

Unit I: Introduction to Object oriented programming

Unit Contains

1.0 Introduction
1.1 Objective
1.2 Object Oriented Programming
   1.2.1 Fundamental concepts and features
   1.2.2 OOP Basics
   1.2.3 Concepts of OOP
   1.2.4 Advantages of OOP
1.3 Object and Classes
   3.3.1 Objects
   3.3.2 Class
1.4 Terms Used in OOP
1.5 Object Linking and Embedding (OLE)
   3.5.1 History of OLE
   3.5.2 Technical
   3.5.3 Criticism
   3.5.4 Interoperability
1.6 Component object model
   3.6.1 Purpose
   3.6.2 Run Time Requirments
   3.6.3 ‘COM’ as an object framework
1.7 Creating Object Variable
1.8 Creating Control Array
1.9 Detecting Control Array
1.10 Let Us Sum Up
1.11 Check Your Progress: The Key
1.12 References
1.0 Introduction

Generally, Object-oriented programming (OOP) is a programming paradigm that uses ‘objects’ i.e. data structures consisting of data fields and methods together with their interactions to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, modularity, polymorphism, and inheritance. A major factor in the invention of Object-Oriented approach is to remove some of the flaws encountered with the procedural approach. In OOP, data is treated as a critical element and does not allow it to flow freely. It bounds closely to the functions that operate on it and protects it from accidental modification from outside functions. OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects. A major advantage of OOP is code reusability. This Unit provides a general introduction to the OOP concepts, major terms used in OOP, OLE and Component Object Model etc.

1.1 Objective

The main aim of this unit is to understand basic features of an Object Oriented Programming and how Visual Basic is used in reference to the Object Oriented Paradigm. After going through this unit you should be able to:
- Understand the Visual Basic and Concept of OOPs
- Understand how to design and implement programs in Visual Basic.
- Learn about Visual Basic as well as basic terms which are used in Visual Basic like languages.

1.2 Object oriented programming

OOP is generally based on the following concepts,

1. **Class** is a user-defined datatype that contains the variables, properties and methods. A class defines the abstract characteristics of a thing (object), including its characteristics (its attributes, fields or properties) and the thing’s behaviors (the methods, operations or features). We can also say that a class is a blueprint or factory that describes the nature of something. For example, the class Dog would consist of traits shared by all dogs, such as breed and fur color (characteristics), and the ability to bark and sit (behaviors). Classes provide modularity and structure in an object-oriented computer program. A class should typically be recognizable to a non-programmer familiar with the problem domain, meaning that the characteristics of the class should make sense in context. Also, the code for a class should be relatively self-contained (generally using encapsulation). Collectively, the properties and methods defined by a class are called members.

2. **Object Linking and Embedding (OLE)** is a technology developed by Microsoft Corporation and it allows embedding and linking to documents and other objects. For developers, it brought OLE Control eXtension (OCX), a way to develop and use custom user interface elements. On a technical level, an OLE object is any object that implements the OLEObject interface, possibly along with a wide range
of other interfaces, depending on the object's needs. OLE allows an
editor to 'farm out' part of a document to another editor and then re-
import it.
For example, a desktop publishing system might send some text to a
word processor or a picture to a bitmap editor using OLE. The main
benefit of using OLE is to display visualizations of data from other
programs that the host program is not normally able to generate itself
(e.g. a pie-chart in a text document), as well as to create a master file.
References to data in this file can be made and the master file can then
have changed data which will then take effect in the referenced
document. This is called 'linking'.

Its primary use is for managing compound documents, but it is also
used for transferring data between different applications using drag and
drop and clipboard operations. The concept of 'embedding' is also
central to much use of multimedia in Web pages, which tend to embed
video, animation (including Flash animations), and audio files within the
hypertext markup language or other structural markup language used
possibly, but not necessarily, using a different embedding mechanism
than OLE.

3. Component Object Model (COM), the fundamental principles of COM
have their roots in Object-Oriented philosophies. It is a platform for the
realization of Object-Oriented Development and Deployment. Because
COM is a runtime framework, types have to be individually identifiable
and specifiable at runtime. To achieve this, globally unique identifiers
(GUIDs) are used. Each COM type is designated its own GUID for
identification at runtime (versus compile time).

1.2.1 Fundamental concepts and features

A survey by Deborah J. Armstrong of nearly 40 years of computing
literature identified a number of fundamental concepts, found in the strong
majority of definitions of OOP. Not all of these concepts are to be found in all
object-oriented programming languages, and so object-oriented programming
that uses classes is sometimes called class-based programming. In particular,
prototype-based programming does not typically use classes. As a result, a
significantly different yet analogous terminology is used to define the concepts
of object and instance.
Benjamin Ciure Pierce and some other researchers view as futile any attempt
to distill OOP to a minimal set of features. He nonetheless identifies
fundamental features that support the OOP programming style in most object-
oriented languages:

- Dynamic dispatch – when a method is invoked on an object, the object
  itself determines what code gets executed by looking up the method at
  run time in a table associated with the object. This feature distinguishes
  an object from an abstract data type (or module), which has a fixed
  (static) implementation of the operations for all instances. It is a
  programming methodology that gives modular component development
  while at the same time being very efficient.
- **Encapsulation** (or multi-methods, in which case the state is kept separate)
- **Subtype polymorphism**
- **Object inheritance** (or delegation)
- **Open recursion** – a special variable (syntactically it may be a keyword), usually called this or self, that allows a method body to invoke another method body of the same object. This variable is late-bound; it allows a method defined in one class to invoke another method that is defined later, in some subclass thereof. John C. Mitchell identifies four main features of programming languages: dynamic dispatch, abstraction, subtype polymorphism, and inheritance.

A **Class** statement defines a new data type. A class is a fundamental building block of object-oriented programming (OOP). The declaration context for a class must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure or block.

Each instance of a class has a lifetime independent of all other instances. This lifetime begins when it is created by a `New (Visual Basic)` clause or by a function such as `CreateObject Function (Visual Basic)`. It ends when all variables pointing to the instance have been set to `Nothing (Visual Basic)` or to instances of other classes. Classes default to `Friend (Visual Basic)` access. One can adjust their access levels with the access modifiers.

### 1.2.2 OOP Basics

Visual Basic is Object-Based, Visual Basic .NET is Object-Oriented, which means that it's a true Object-Oriented Programming Language. Visual Basic .NET supports all the key OOP features like Polymorphism, Inheritance, Abstraction and Encapsulation.

A major factor in the invention of Object-Oriented approach is to remove some of the flaws encountered with the procedural approach.

Some important features of Object Oriented programming are as follows:
- Emphasis on data rather than procedure;
- Programs are divided into Objects;
- Data is hidden and cannot be accessed by external functions;
- Objects can communicate with each other through functions;
- New data and functions can be easily added whenever necessary;
- Follows bottom-up approach.

### 1.2.3 Concepts of OOP

Some of the basic concepts of OOP are listed below,
- Objects
- Classes
- Data Abstraction and Encapsulation
- Inheritance
- Polymorphism

**Objects**: Objects are the basic run-time entities in an object-oriented system. Programming problem is analyzed in terms of objects and nature of communication between them. When a program is executed, objects interact
with each other by sending messages. Different objects can also interact with each other without knowing the details of their data or code.

**Classes:** A class is a collection of objects of similar type. Once a class is defined, any number of objects can be created which belong to that class.

**Data Abstraction and Encapsulation:** Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes. Storing data and functions in a single unit (class) is encapsulation. Data cannot be accessible to the outside world and only those functions which are stored in the class can access it.

**Inheritance:** Inheritance is the process by which objects can acquire the properties of objects of other class. In OOP, inheritance provides reusability, like, adding additional features to an existing class without modifying it. This is achieved by deriving a new class from the existing one. The new class will have combined features of both the classes.

**Polymorphism:** Polymorphism means the ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends on the data types used in the operation. Polymorphism is extensively used in implementing Inheritance.

### 1.2.4 Advantages of OOP

Object-Oriented Programming has the following advantages over conventional approaches:

- OOP provides a clear modular structure for programs which makes it good for defining abstract datatypes where implementation details are hidden and the unit has a clearly defined interface.
- OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
- OOP provides a good framework for code libraries where supplied software components can be easily adapted and modified by the programmer. This is particularly useful for developing graphical user interfaces.

### 1.3 Object and Classes

#### 1.3.1 Objects

We have already discussed in section 3.2.3 that Objects are the basic run-time entities in an object-oriented system. Here we can say that a programming problem can be analyzed in terms of objects and nature of communication between them. When a program is executed, objects interact with each other by sending messages. Even different objects can also interact with each other without knowing the details of their data. Objects are the central idea behind OOP. The idea is quite simple:

An object is a bundle of variables and related methods.
A method is similar to a procedure. The basic idea behind an object is that of simulation. Most programs are written with very little reference to the real world objects the program is designed to work with; in object oriented methodology, a program should be written to simulate the states and activities of real world objects. This means that apart from looking at data structures when modelling an object, we must also look at methods associated with that object, in other words, functions that modify the objects attributes.

1.3.2 Class
As we state earlier, a Class is a user-defined datatype that contains the variables, properties and methods in it. A class defines the abstract characteristics of a thing (object), including its attributes, fields or properties and the things it can do, or methods, operations or features. Classes provide modularity and structure in an object-oriented computer program. A class should typically be recognizable to a non-programmer familiar with the problem domain, meaning that the characteristics of the class should make sense in context. Also, the code for a class should be relatively self-contained (generally using encapsulation). Collectively, the properties and methods defined by a class are called members.

One can have an instance of a class; the instance is the actual object created at run-time. In programmer vernacular, the Lassie object is an instance of the Dog class. The set of values of the attributes of a particular object is called its state. The object consists of state and the behavior that's defined in the object's classes.

```
[<attributelist>] [accessmodifier] [Shadows] [MustInherit | NotInheritable]
[Partial] _
Class name [(Of typelist)]
    [Inherits classname]
    [Implements interfacenames]
    [statements]
End Class
```

1.3.2.1 Rules for Classes

- **Nesting:** One can define one class within another. The outer class is called the containing class, and the inner class is called a nested class.
- **Inheritance:** If the class uses the Inherits Statement, one can specify only one base class or interface. A class cannot inherit from more than one element. A class cannot inherit from another class with a more restrictive access level. For example, a Public class cannot inherit from a Friend class. A class cannot inherit from a class nested within it.
- **Implementation:** If the class uses the Implements Statement, you must implement every member defined by every interface you specify in interface-names. An exception to this is reimplementation of a base class member.
- **Default Property:** A class can specify at most one property as its default property.
- **Access Level.** Within a class, one can declare each member with its own access level. Class members default to Public (Visual Basic) access, except variables and constants, which default to Private (Visual Basic) access. When a class has more restricted access than one of its members, the class access level takes precedence.

- **Scope:** A class is in scope throughout its containing namespace, class, structure, or module. The scope of every class member is the entire class.

- **Lifetime:** Visual Basic does not support static classes. The functional equivalent of a static class is provided by a module. Class members have lifetimes depending on how and where they are declared.

- **Qualification:** Code outside a class must qualify a member’s name with the name of that class.

If code inside a nested class makes an unqualified reference to a programming element, Visual Basic searches for the element first in the nested class, then in its containing class, and so on out to the outermost containing element.

---

### Check Your Progress 1

**Notes:**

i) Write your answers in the space given below.

ii) Compare Your Answer with those given at the end of the unit.

a) **What is Object Oriented Programming? Give major features of OOP?**

   **Ans.** .................................................................................................................................
   .................................................................................................................................
   .................................................................................................................................
   .................................................................................................................................

b) **Differentiate between ‘Class’ and ‘Object’. Give the advantages of OOP.**

   **Ans.** .................................................................................................................................
   .................................................................................................................................
   .................................................................................................................................
   .................................................................................................................................

---

### 1.4 Terms used in OOPS

**Class**

A Class is a user-defined datatype that contains the variables, properties and methods in it.

**Instance**

One can have an instance of a class; the instance is the actual object created at run-time. In programmer view, the Lassie object is an instance of the Dog class. The set of values of the attributes of a particular object is called its *state*. The object consists of state and the behavior that's defined in the object's classes.
Method
Method is a set of procedural statements for achieving the desired result. It performs different kinds of operations on different data types. In a programming language, methods (sometimes referred to as ‘functions’) are verbs.

Message Passing
The process by which an object sends data to another object or asks the other object to invoke a method. Also known to some programming languages as interfacing.

Inheritance
Inheritance is a process in which a class inherits all the state and behavior of another class. This type of relationship is called child-Parent or is-a relationship. “Subclasses” are more specialized versions of a class, which inherit attributes and behaviors from their parent classes, and can introduce their own. Multiple inheritance is inheritance from more than one ancestor class, neither of these ancestors being an ancestor of the other.

Abstraction
Abstraction is simplifying complex reality by modeling classes appropriate to the problem, and working at the most appropriate level of inheritance for a given aspect of the problem.

Encapsulation
Encapsulation conceals the functional details of a class from objects that send messages to it.

Polymorphism
Polymorphism allows the programmer to treat derived class members just like their parent class’s members. More precisely, Polymorphism in object-oriented programming is the ability of objects belonging to different data types to respond to calls of methods of the same name, each one according to an appropriate type-specific behavior. One method, or an operator such as +, -, or *, can be abstractly applied in many different situations.

Decoupling
Decoupling allows for the separation of object interactions from classes and inheritance into distinct layers of abstraction. A common use of decoupling is to polymorphically decouple the encapsulation which is the practice of using reusable code to prevent discrete code modules from interacting with each other. However, in practice decoupling often involves trade-offs with regard to which patterns of change to favor. The science of measuring these trade-offs in respect to actual change in an objective way is still in its infancy.

1.5 Object Linking and Embedding
As discussed above Object Linking and Embedding (OLE) is a technology developed by Microsoft that allows embedding and linking to documents and other objects.
1.5.1 History of OLE

OLE 1.0, released in 1990, was the evolution of the original dynamic data exchange, or DDE, concepts that Microsoft developed for earlier versions of Windows. While DDE was limited to transferring limited amounts of data between two running applications, OLE was capable of maintaining active links between two documents or even embedding one type of document within another. OLE servers and clients communicate with system libraries using virtual function tables, or VTBLs. The VTBL consists of a structure of function pointers that the system library can use to communicate with the server or client. The server and client libraries, OLESVR.DLL and OLECLI.DLL, were originally designed to communicate between themselves using the WM_DDE_EXECUTE message. OLE 1.0 later evolved to become an architecture for software components known as the Component Object Model (COM), and later DCOM.

When an OLE object is placed on the clipboard or embedded in a document, both a visual representation in native Windows formats (such as a bitmap or metafile) is stored, as well as the underlying data in its own format. This allows applications to display the object without loading the application used to create the object, while also allowing the object to be edited, if the appropriate application is installed. The Object Packager, a component of OLE, shipping from Windows 3.1 up to Windows XP allows a non-OLE object to be ‘packaged’ so it can be embedded into an OLE client.

OLE 2.0 was the next evolution of OLE 1.0, sharing many of the same goals, but was re-implemented on top of the Component Object Model instead of using VTBLs directly. New features were automation, drag-and-drop, in-place activation and structured storage. Monikers evolved from OLE 1 object names, and provided an hierarchical object and resource naming system similar to URLs or URIs, which were independently invented. Windows now has merged the two technologies supporting a URL Moniker type, and a Moniker URL scheme. Today Microsoft's Internet Explorer uses the operating system's URL Monikers internally to download resources. Applications which do so can share IE's cache. However most (if not all) competing browsers have a completely independent implementation of Web protocols, including their own cache, and do not use the Moniker system.

OLE custom controls were introduced in 1994 as a replacement for the now deprecated Visual Basic Extension controls. Instead of upgrading these, the new architecture was based on OLE. In particular, any container that supported OLE 2.0 could already embed OLE custom controls, although these controls cannot react to events unless the container supports this. OLE custom controls are usually shipped in the form of a dynamic link library with the .ocx extension. In 1996 all interfaces for controls (except IUnknown) were made optional to keep the file size of controls down, so they would download faster; these were then called ActiveX Controls.
1.5.2 Technical Details

OLE objects and containers are implemented on top of the Component Object Model; they are objects that can implement interfaces to export their functionality. Only the OleObject interface is compulsory, but other interfaces may need to be implemented as well if the functionality exported by those interfaces is required. To ease understanding of what follows, a bit of terminology has to be explained.

**OLE objects**

- **DataObject**: When implemented, enables the transfer of data, and notification of data changes. It must be implemented by objects that are to support drag-and-drop, being copied to or pasted from the clipboard, or being linked or embedded in a containing document.

- **ObjectWithSite**: Allows the caller to inform the OLE object of its site. This functionality is also provided by OleObject, but ObjectWithSite can be used, when supported, if OleObject is not used for other matters.

- **OleCache**: Allows visual presentations from a DataObject to be cached. This allows an embedded object to store its visual representation, thus enabling it to be displayed later without needing to start the application that was used to create the object.

- **OleCacheControl**: This interface is not called by the container, but internally by the object to allow it to receive notifications of when its DataObject is running, thereby allowing it to subscribe to notifications of data changes of that object and thus allowing it to update the cached presentation properly.

- **OleDocument**: Allows the OLE object to support multiple views of its data, as well as a few related functions.

- **OleDocumentView**: A document object (an object that implements OleDocument) implements this interface for every view. It allows the caller to set the site of the object, query and set the size of the object and to show and activate it, as well as some related functions.

- **OleWindow**: A windowless object is an object that doesn't have its own window but it instead displayed in its container's window. It is used by the container to relay messages received by the container's window that are intended for the contained object.
- **OleLink**: Allows the object to support linking, e.g. by allowing the container to set the source of a linked object.
- **OleObject**: Arguably the most important interface for an OLE object. For example, it allows the container to inform the object of its site, initialize the object from data, to open and close it, to query and set the size of the object, to ask for notifications on the container's AdviseSink and to execute objects defined as 'verbs' on the object. These verbs often include 'Open' or 'Edit', but can also include other verbs. One of the verbs is defined to be the principal verb, and it is executed when the user double-clicks an object.
- **ViewObject**: Allows an object to draw itself directly, without passing a DataObject to the container. For objects that support both DataObject and this interface, the underlying implementation is usually shared.

**OLE container**

- **adviseSink**: Allows the implementer to receive notifications when the object is saved, closed, renamed, or when its data or visual presentation changes.
- **oleClientSite**: This interface allows the caller to obtain information on the container and location of an object, as well requesting that the object be saved, resized, shown, hidden, etc.
- **oleDocumentSite**: Allows the caller to ask for the object on this site to be activated immediately. If this interface is implemented, OleClientSite, OleInPlaceSite and AdviseSink must be implemented as well.
- **oleContainer**: This interface allows the caller to enumerate embedded objects in a container, or to find such objects by name. It is primarily useful if the container wishes to support links to embedded objects.
- **oleWindow**: Allows the caller to ask the container to insert its menu items in an empty menu that will become the cooperative menu. Also allows the caller to ask the container to show or hide this menu, to show or hide dialog boxes, and to process accelerator keys received by the contained object intended for the container.
- **OleUILinkContainer**: Contains the methods that the standard OLE dialog boxes that manage linked objects use to update linked objects in a container, or to query and change their sources. Used by the "Links", "Change source", "Update links" and "Object properties" dialog boxes.
- **oleUndoManager**: Provides a centralized undo service to both the container itself and to embedded objects. When an undoable action is performed, an IOleUndoUnit is created and added to the IOleUndoManager.

### 1.5.3 Criticism

There was a competing technology, [OpenDoc](https://en.wikipedia.org/wiki/OpenDoc), that was widely considered to be both easier to use and more robust than OLE. OpenDoc allowed users to view and edit information across applications, directly in competition with Microsoft's proprietary OLE standard. A consortium called the Component Integration Laboratories (CIL) was established in 1993 by some Microsoft competitors to create OpenDoc as an 'open-source' standard for cross-platform linking and embedding. Microsoft unilaterally announced that its OLE proprietary technology would be incorporated directly into MS Windows operating system.
Microsoft then required OLE compatibility as a condition of Microsoft's certification of an application's compatibility with MS Windows 95. Microsoft initially announced that applications using OpenDoc would be deemed compatible with OLE, and would receive certification for Windows 95, because OpenDoc was a 'superset' of OLE and it provided every function of OLE, and more. Microsoft later announced that applications using OpenDoc would not receive automatic certification, and might not receive certification at all.

1.5.4 Inteoperability
Use of Microsoft Object Linking and Embedding (OLE) objects limits the interoperability, because these objects are not widely supported in programs for viewing or editing files (e.g. embedding of other files inside the file, such as tables or charts from a spreadsheet application in a text document or presentation file) If software that understands an OLE object is not available, the object is usually replaced by a picture (bitmap representation of the object) or not displayed at all.

1.6 Component Object Model (COM)

1.6.1 Purpose
COM is platform-independent, distributed, object-oriented system for creating binary software components that can interact. COM is the foundation technology for Microsoft's OLE (compound documents) and ActiveX (Internet-enabled components) technologies.

COM objects can be created with a variety of programming languages. Object-oriented languages, such as C++, provide programming mechanisms that simplify the implementation of COM objects. These objects can be within a single process, in other processes or even on remote computers.

1.6.2 Run Time requirements
Before COM, the whole technology was named Object Linking and Embedding (OLE).
COM is only a subset of OLE. OLE2 introduced features like Automation and Controls, and all the inner subsets of the technology were given a distinct name (Structured Storage, Compound document, Property). ActiveX was used to cover the whole technology. OLE Automation became Automation, and the ActiveX keyword was the only name to represent the future in composition and design of object-oriented applications, previously named OLE. OLE was available for 16-bits platforms, and was the first to be available from Visual Basic 3.0 with VBX controls. OLE2 and the 32bit architecture replaced this, the controls became OLE Controls, and Visual Basic 4.0 and 5.0 allowed VB to create real OLE Controls. COM is a popular Develop Mentor expression.

A whole range of Microsoft Server product and inner functionalities of the Windows OS are designed using COM. COM relies on Microsoft RPC and LRPC (memory). Implementation and execution are extremely fast because a COM component is the exact binary representation of an abstract C++ class with its implementation classes, and a registry entry to link the name of the component with its hosted modules (DLL or EXE).
COM is the longest technology used by Microsoft, either into its products or into its range of operating systems. Every product exposes its functionalities via an SDK and most of them have COM API. Office object model are OLE2 object model, that is COM Windows Management Console (MMC snap-ins) and are COM components. Inserting a Word Object into an Excel Workbook depends on COM. In 2007, COM is still a competitor for .NET. Enterprises services and COM services components are still in place. .NET and COM have a fully interpose paradigm. COM is the fundamental piece of technology that is shipped in every Windows.

1.6.3 ‘COM’ As an Object Framework
The fundamental principles of COM have their roots in Object-Oriented philosophies. It is a platform for the realization of Object-Oriented Development and Deployment. Because COM is a runtime framework, types have to be individually identifiable and specifiable at runtime. To achieve this, globally unique identifiers (GUIDs) are used. Each COM type is designated its own GUID for identification at runtime (versus compile time). In order for information on COM types to be accessible at both compile time and runtime, COM uses type libraries. It is through the effective use of type libraries that COM achieves its capabilities as a dynamic framework for the interaction of objects.

Consider the following example coclass definition in an IDL:

coclass MyObject
{
    [default] interface IMyObject;
    [default, source] dispinterface _IMyObjectEvents;
};

The above code fragment declares a COM class named MyObject which must implement an interface named IMyObject and which supports (not implements) the event interface _IMyObjectEvents.

Ignoring the event interface bit, this is conceptually equivalent to defining a C++ class like this:

class CSomeObject : public ISomeInterface
{
    ...
    ...
};
where ISomeInterface is a C++ pure virtual class.
Check Your Progress 2

Notes: i) Write your answers in the space given below.
    ii) Compare Your Answer with those given at the end of the unit.

a) What do you understand by ‘Object Linking and Embedding’?
   Ans. ..........................................................................................................................
   ..........................................................................................................................
   ..........................................................................................................................
   ..........................................................................................................................

b) What is COM? Explain in brief.
   Ans. ..........................................................................................................................
   ..........................................................................................................................
   ..........................................................................................................................
   ..........................................................................................................................

1.7 Creating Object Variable in VB
You can assign an object to a variable for the same reasons you assign any value to a variable:

- A variable name is often shorter and easier to remember than the full path of methods and properties necessary to access the object itself.
- Using a variable that refers to an object is more efficient than repeatedly accessing the object itself through the necessary methods or properties.
- You can change a variable to refer to other objects while your code is running.

You can use object variables to shorten the code you have to type. The following example uses the full path of methods and properties to access a Control object.

```vbnet
' Assume Me is a valid Form, or replace Me with a valid Form.
Me ActiveForm.ActiveControl.Text = "Test"
Me ActiveForm.ActiveControl.Location = New Point(100, 100)
Me ActiveForm.ActiveControl.Show()
```

You can shorten this code, and speed up execution, if you use an object variable for the control. You should declare the object variable with the specific class that you intend to assign to it (Control in this case). Once you assign an object to the variable, you can treat it exactly the same as you treat the object to which it refers. You can set or retrieve the properties of the object or use any of its methods. The following example uses an object variable to simplify the code in the preceding example.

```vbnet
Dim ctrlActv As System.Windows.Forms.Control =
Me ActiveForm.ActiveControl
ctrlActv.Text = "Test"
ctrlActv.Location = New Point(100, 100)
ctrlActv.Show()
```
A control array is a group of controls that share the same name type and the same event procedures. Adding controls with control arrays uses fewer resources than adding multiple control of same type at design time.

A control array can be created only at design time, and at least one control must belong to it. You can create a control array by following one of these three methods:

- You can create a control and then assign a numeric, non-negative value to its Index property; you have thus created a control array with just one element.
- You create two controls of the same class and assign them an identical Name property. Visual Basic shows a dialog box warns you that there's already a control with that name and asks whether you want to create a control array. Click on the Yes button.
- You can select a control on the form, press Ctrl+C to copy it to the clipboard, and then press Ctrl+V to paste a new instance of the control, which has the same Name property as the original one. Visual Basic shows the warning mentioned in the previous bullet.

Control arrays are one of the most interesting features of the Visual Basic environment, and they add a lot of flexibility to your programs:

- Controls that belong to the same control array share the same set of event procedures; this often dramatically reduces the amount of code you have to write to respond to a user's actions.
- You can dynamically add new elements to a control array at run time; in other words, you can effectively create new controls that didn't exist at design time.
- Elements of control arrays consume fewer resources than regular controls and tend to produce smaller executables. Visual Basic forms can host up to 256 different control names, but a control array counts as one against this number. In other words, control arrays let you effectively overcome this limit.

The importance of using control arrays as a means of dynamically creating new controls at run time is somewhat reduced in Visual Basic 6, which has introduced a new and more powerful capability.

Don't let the term array lead you to think control array is related to VBA arrays; they're completely different objects. Control arrays can only be one-dimensional. They don't need to be dimensioned: Each control you add automatically extends the array. The Index property identifies the position of each control in the control array it belongs to, but it's possible for a control array to have holes in the index sequence. The lowest possible value for the Index property is 0. You reference a control belonging to a control array as you would reference a standard array item:

```
Text1(0).Text = ""
```

Sharing Event Handlers
The following example demonstrates sharing the Change event handler for a group of three TextBox controls. The Handles clause of the event handler specifies which control the event will handle. The event handler returns a generic Object, so it must be cast to the specific object type (in this case, TextBox) that you want to handle using the DirectCast method.

```vba
Private Sub Text1_Change(Index As Integer)
    Select Case Index
    Case 0
        MsgBox(“The text in the first TextBox has changed“)
    Case 1
        MsgBox(“The text in the second TextBox has changed“)
    Case 2
        MsgBox(“The text in the third TextBox has changed“)
    End Select
End Sub
```

1.8 Creating Control Array

To create a control at run time using this method you first create a control array for the control you wish to dynamically create. In other words, if you want to create checkboxes at run time, you first must create a control array of checkboxes. For those of you unfamiliar with the term, a control array is a collection of controls on a form, all having the same name, and possessing unique Index property values. It's possible to create a Control array that has just a single 'member' and that's what we'll do now.

You can begin by placing a Checkbox and a Command button on a form.

In the Command Button, it'll be placing code to dynamically create a checkbox on the form at run time. First, you need to tell Visual Basic that the Checkbox is a member of a Control Array you do that merely by changing its Index property from the default blank value to a number in this case 0.
Once the Index property has been set to 0, the Checkbox is now a member of the Check1 Checkbox Control Array which makes creating a new checkbox at runtime very easy.

All we need to do is tell Visual Basic that we want to create a new checkbox, using the existing Checkbox as a template. We do this by executing the Visual Basic Load Statement within the Click Event Procedure of the Command Button like this.

```vba
Private Sub Command1_Click()
    Load Check1(1)
    Check1(1).Caption = "New Checkbox"
End Sub
```

The Load statement Load Check1(1), tells Visual Basic to create a new member of the Check1 checkbox array and to create it with an Index property of 1. This statement

Check1(1).Caption = "New Checkbox"

gives the Checkbox, whose index property is equal to 1, a unique caption to make it 'stand out' form the original checkbox placed on the form at design time.

If we now run the program, and then click on the Command Button we’ll see this screenshot
This gives wrong results because when you create a control at runtime, by
default, the Visible property of the new control is set to False. To see the new
control, we need to explicitly set its Visible property to True. Let's modify the
code in the Click event procedure to look like this,

Private Sub Command1_Click()
Load Check1(1)
Check1(1).Caption = "New Checkbox"
Check1(1).Visible = True
End Sub
Now let's run the program again, and click on the Command Button once
more…

Something is still wrong, there's still just the single checkbox!

The problem is this: when the new control is created, the properties of the new
control are identical to the properties of the 'template' control used to create it,
with the exception of the Index property which we set with the Load statement,
and the visible property which we know is initialized to False. Because of that,
the new checkbox is on the form, it just so happens to be sitting 'under' the first
control, since it has identical Top, Left, Height and Width properties.

All we need to do to see the new control is to move it away from the first
control and we can do that by adding a line of code to the Click Event
procedure to change the Top property of the new control. Like this,
Private Sub Command1_Click()
Load Check1(1)
Check1(1).Caption = "New Checkbox"
Check1(1).Visible = True
Check1(1).Top = Check1(0).Top + Check1(0).Height
End Sub

The line of code `Check1(1).Top = Check1(0).Top + Check1(0).Height` tells VB to take the current value of the Top property of the existing checkbox, and to add to that the value of its Height Property (remember, the first control has an Index property of 0). The result of this addition is a Top property for the new control that is just under the first control.

If we now run the program, and click on the Command Button, we'll see this screen shot,

Now it works.

To review, here's a summary of the steps necessary to create a new control using the Control Array method.

a) Create a control array of the control type you wish to create at runtime. If you want to create a textbox at runtime, create a Textbox control array. If you want to create a Command Button, create a Command Button Control array. Remember, to create a Control Array, all you need to do is to change the Index property of the control to something other than its default empty value.

b) Use the Load Statement, with a unique Index property, to create the new control.

c) Change the Visible property of the new control to True in order to make it visible.

d) Change the coordinate properties (Left or Top) to bring the new control out from under the original.
1.9 Detecting Control Array

We have a number of controls on a form, some of which may be part of a control array. We have a generic bit of code which is building up information on each control it encounters. We store away the name of the control, but also need to identify whether a particular control is part of an array. We can determine this if the Index property is set, however code fails if it discovers a control not part of an array e.g.

```
If IsNumeric(ControlOnForm.index) Then
    .ActualControlKey = ControlOnForm.Name
Else
    .ActualControlKey = ControlOnForm.Name & "(" & ControlOnForm.index & ")"
End If
```

1.10 Let us Sum Up

Object-oriented programming (OOP) is a programming paradigm that uses ‘objects’ i.e. data structures consisting of data fields and methods together with their interactions to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, modularity, polymorphism, and inheritance. Class is a user-defined datatype that contains the variables, properties and methods in it. OLE is a technology developed by Microsoft Corporation and it allows embedding and linking to documents and other objects. COM is a platform for the realization of Object-Oriented Development and Deployment. Because COM is a runtime framework, types have to be individually identifiable and specifiable at runtime. A major advantage of OOP is code reusability. Objects are the basic run-time entities in an object-oriented system. Abstraction refers to the act of representing essential features without including the background details or explanations. Storing data and functions in a single unit (class) is

Check Your Progress 3

Notes: i) Write your answers in the space given below.
    ii) Compare Your Answer with those given at the end of the unit.

a) What is Control Array? What are the methods for creation of a Control Array?
   Ans: ........................................................................................................................................
   ........................................................................................................................................
   ........................................................................................................................................
   ........................................................................................................................................

b) Give all steps necessary to create a new control using the Control Array Method in VB.
   Ans: ........................................................................................................................................
   ........................................................................................................................................
   ........................................................................................................................................
   ........................................................................................................................................
**encapsulation. Inheritance** is the process by which objects can acquire the properties of objects of other class. **Polymorphism** means the ability to take more than one form. One of the main advantages of OOP is that it provides a clear modular structure for programs which makes it good for defining abstract datatypes where implementation details are hidden and the unit has a clearly defined interface.

**Object Linking and Embedding** (OLE) is a technology developed by Microsoft that allows embedding and linking to documents and other objects. Use of Microsoft Object Linking and Embedding (OLE) objects limits the interoperability, because these objects are not widely supported in programs for viewing or editing files.

**COM** is a platform-independent, distributed, object-oriented system for creating binary software components that can interact. COM is the foundation technology for Microsoft's OLE (compound documents) and ActiveX (Internet-enabled components) technologies.

### 1.11 Check Your Progress: The Key

**Check Your Progress 1**

**a) Object-oriented programming** (OOP) is a **programming paradigm** that uses ‘objects’ i.e data structures consisting of data fields and methods together with their interactions to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, modularity, polymorphism, and inheritance. Some important features of Object Oriented programming are as follows:

- Emphasis on data rather than procedure;
- Programs are divided into Objects;
- Data is hidden and cannot be accessed by external functions;
- Objects can communicate with each other through functions;
- New data and functions can be easily added whenever necessary;
- Follows bottom-up approach.

**b) Objects** are the basic run-time entities in an object-oriented system. Programming problem is analyzed in terms of objects and nature of communication between them. When a program is executed, objects interact with each other by sending messages. Different objects can also interact with each other without knowing the details of their data or code. On the other hand, a **Class** is a collection of objects of similar type. Once a class is defined, any number of objects can be created which belong to that class. Object-Oriented Programming has the following advantages over conventional approaches:

- OOP provides a clear modular structure for programs which makes it good for defining abstract datatypes where implementation details are hidden and the unit has a clearly defined interface.
- OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
- OOP provides a good framework for code libraries where supplied software components can be easily adapted and modified by the
programmer. This is particularly useful for developing graphical user interfaces.

Check Your Progress 2

a) Object Linking and Embedding (OLE) is a technology developed by Microsoft that allows embedding and linking to documents and other objects. For developers, it brought OLE Control eXtension (OCX), a way to develop and use custom user interface elements. On a technical level, an OLE object is any object that implements the IOleObject interface, possibly along with a wide range of other interfaces, depending on the object's needs. OLE allows an editor to 'farm out' part of a document to another editor and then re-import it. The main benefit of using OLE is to display visualizations of data from other programs that the host program is not normally able to generate itself, as well as to create a master file. References to data in this file can be made and the master file can then have changed data which will then take effect in the referenced document. This is called 'linking' (instead of 'embedding'). Its primary use is for managing compound documents, but it is also used for transferring data between different applications using drag and drop and clipboard operations. The concept of 'embedding' is also central to much use of multimedia in Web pages, which tend to embed video, animation, and audio files within the hypertext markup language or other structural markup language used possibly, but not necessarily, using a different embedding mechanism than OLE.

b) COM is a platform-independent, distributed, object-oriented system for creating binary software components that can interact. COM is the foundation technology for Microsoft's OLE (compound documents) and ActiveX (Internet-enabled components) technologies. COM objects can be created with a variety of programming languages. Object-oriented languages, such as C++, provide programming mechanisms that simplify the implementation of COM objects. These objects can be within a single process, in other processes, even on remote computers.

Check Your Progress 3

a) Control arrays are one of the most interesting features of the Visual Basic environment. A control array is a group of controls that share the same name type and the same event procedures. A control array can be created only at design time, and at the very minimum at least one control must belong to it. You create a control array following one of these three methods:

- You create a control and then assign a numeric, non-negative value to its Index property; you have thus created a control array with just one element.
- You create two controls of the same class and assign them an identical Name property. Visual Basic shows a dialog box warning you that there's already a control with that name and asks whether you want to create a control array. Click on the Yes button.
You select a control on the form, press Ctrl+C to copy it to the clipboard, and then press Ctrl+V to paste a new instance of the control, which has the same Name property as the original one. Visual Basic shows the warning mentioned in the previous bullet.

b) Following are the steps necessary to create a new control using the Control Array method,

- Create a control array of the control type you wish to create at runtime. If you want to create a textbox at runtime, create a Textbox control array. If you want to create a Command Button, create a Command Button Control array. Remember, to create a Control Array, all you need to do is to change the Index property of the control to something other than its default empty value.
- Use the Load Statement, with a unique Index property, to create the new control.
- Change the Visible property of the new control to True in order to make it visible.
- Change the coordinate properties (Left or Top) to bring the new control out from under the original.

1.12 References

4. Advanced Microsoft Visual Basic 6.0 by the Mandelbrot Set, MS Press, ISBN 1572318937
Unit 2: Basics of Form

2.0 Introduction
In this unit, student will learn about what is form in Visual Basic, How to work with form, what are the components of Form, How it is created and linked with another form. Also one can create an application with the help of different form tools.

2.1 Objective
The form object is essentially a container for holding the controls that allow the user to interact with an application. In VB we will look at many options for configuring a windows form, creating form, change its properties, tools and controlling one form with another. We will also look briefly about MDI form, adding controls to a form and we will take a much closer look at the steps involved in laying out controls on a form in VB. We will also learn about basic terminologies of form window, form layout window and MDI forms etc.

2.2 Concept of form, Its Importance and Properties:-

Concept
A form is nothing but a box that contains control objects. A form represents a window or dialog box visual appearance. A form is actually a Visual basic component with its own set of properties and events. In Visual Basic, the form is the container for all the controls that make up the user interface. When a Visual Basic application will executing, each window displays on the desktop is a Form.

Form is a container on which we placed the elements of the user interface.
If this is the first time you have used Visual Basic since its installation, the Visual Basic window will probably look some thing like this:

Fig 1: basic window of VB

The Appearance of Forms:

The main characteristic of a form is the title bar on which the form’s caption is displayed. One of the left ends of the title bar is the control menu & on the right side of the title bar are three buttons: minimize, maximize and close. The space below the title bar is used to user interface & all the user interface controls are show on this space.
There are two view of Form

THE FORM IT SELF
(The elements of the visible user interface)

THE FORM IT SELF:-
Here's a simple Visual Basic form. It looks just like any other form that we use in Windows applications. The header area has a caption, the control menu, and the minimize/maximize/close buttons. The large area of the form is called the client area.

A CODE WINDOW:
We must add some code behind the Form’s button to perform the action indicated by their caption. If the command button caption is clear then it work for deleting/clear the text. These step are used to open the CODE WINDOW.
When the form appears on the screen double-click on the user interface controls (like command button) visual basic opens the code window that is shown in figure.

To switch between the two views, click the little icons (View Code and View Form) at the top of the Project Explorer.

**Project Explorer**

The Project Explorer Docked on the right side of the screen, just under the toolbar. The Project Explorer as shown in figure serves as a quick reference to the various elements of a project namely form, classes and modules. The entire object that makes up the application is packed in a project. A simple project will typically contain one form, which is a window that is designed as part of a program's interface. It is possible to develop any number of forms for use in a program, although a program may consist of a single form. In addition to forms, the Project Explorer window also lists code modules and classes.

**Form Layout**

The form layout window is in the right lower corner of the vb IDE. It determine the initial position of the form in our application.
Properties of Form:-
We can customize the form using properties. We can change properties at run time & design time but the NAME properties customize only at design time.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Return the name used in code to identify an object</td>
</tr>
<tr>
<td>Appearance</td>
<td>Sets whether or not an object is painted at run time with 3-D effects</td>
</tr>
<tr>
<td>Auto redraw</td>
<td>Sets the output from a graphics method to a persistent bitmap</td>
</tr>
<tr>
<td>Back color</td>
<td>Sets the background color used to display text and graphics in an object</td>
</tr>
<tr>
<td>Borderstyle</td>
<td>Sets the border style for an object</td>
</tr>
<tr>
<td>Caption</td>
<td>Sets the text displayed on an object’s title bar or below an object’s icon</td>
</tr>
<tr>
<td>Clipcontrols</td>
<td>Determines whether graphics methods in paint events repaint an entire object or newly exposed areas.</td>
</tr>
<tr>
<td>Controlbox</td>
<td>Returns a value indicating whether a control-menu box is displayed on a form at run time.</td>
</tr>
<tr>
<td>Drawmode</td>
<td>Sets the appearance of output from graphics methods or a shape or line control</td>
</tr>
<tr>
<td>Drawstyle</td>
<td>Determines the line style for output from graphics methods.</td>
</tr>
<tr>
<td>Drawwidth</td>
<td>Sets the line width for output form graphics methods.</td>
</tr>
<tr>
<td>Enable</td>
<td>Sets a value that determines whether an object can respond to user-generated events.</td>
</tr>
<tr>
<td>Fillcolor</td>
<td>Sets the color used to fill in shapes, circles and boxes.</td>
</tr>
<tr>
<td>Fillstyle</td>
<td>Sets the fill style of a shape</td>
</tr>
<tr>
<td>Font</td>
<td>Return a font object</td>
</tr>
<tr>
<td>Fonttransparent</td>
<td>Sets a value that determines whether background text/graphics on a form, printer or picture box are displayed</td>
</tr>
<tr>
<td>ForeColor</td>
<td>Sets the foreground color used to display text and graphics in an object</td>
</tr>
<tr>
<td>HasDC</td>
<td>Determines whether a unique display context is allocated for the control</td>
</tr>
<tr>
<td>Height</td>
<td>Sets the height of an object</td>
</tr>
<tr>
<td>Helpcontext ID</td>
<td>Specifies the default help file context id for an object</td>
</tr>
<tr>
<td>Icon</td>
<td>Return the icon displayed when a form is minimized at run time.</td>
</tr>
<tr>
<td>Key preview</td>
<td>Sets whether keyboard events for an object are invoked before keyboard events for controls on that object</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Left</td>
<td>Sets the distance between the internal left edge of an object and the left edge of its container.</td>
</tr>
<tr>
<td>Linkmode</td>
<td>Sets the type of link used for a dde conversation and activates the connection.</td>
</tr>
<tr>
<td>Linktopic</td>
<td>Sets the source application and topic for a destination control.</td>
</tr>
<tr>
<td>Maxbutton</td>
<td>Determines whether a form has a maximize button.</td>
</tr>
<tr>
<td>Mdichild</td>
<td>Sets a value that determines if a form is displayed as an mdi child form.</td>
</tr>
<tr>
<td>MinButton</td>
<td>Determines whether a form has a minimize button.</td>
</tr>
<tr>
<td>Mouseicon</td>
<td>Sets a custom mouse icon.</td>
</tr>
<tr>
<td>Mousepointer</td>
<td>Sets the type of mouse pointer displayed when drag over part of an object.</td>
</tr>
<tr>
<td>Moveable</td>
<td>Sets a value that determines whether a form can be moved.</td>
</tr>
<tr>
<td>Negotiatemenus</td>
<td>Determines whether or not a form incorporates the menus from an object on the form’s menu bar</td>
</tr>
<tr>
<td>Oledropmode</td>
<td>Sets whether this object can act as an ole drop target.</td>
</tr>
<tr>
<td>Palette</td>
<td>Sets an image that contains the palette to use on an object when palettemode is set to custom</td>
</tr>
<tr>
<td>Palettemode</td>
<td>Sets a value that determines which palette to use for the controls on a object</td>
</tr>
<tr>
<td>Picture</td>
<td>Sets a graphic to be displayed in a control.</td>
</tr>
<tr>
<td>Righttoleft</td>
<td>Determines text display direction and control visual appearance on a bidirectional system</td>
</tr>
<tr>
<td>Scale height</td>
<td>Sets the number of units for the vertical measurement of an object’s interior</td>
</tr>
<tr>
<td>Scale left</td>
<td>Sets the horizontal coordinates for the left edges of an object.</td>
</tr>
<tr>
<td>Scale mode</td>
<td>Sets a value indicating measurement units for object coordinates when using graphics methods or positioning controls</td>
</tr>
<tr>
<td>Scale top</td>
<td>Sets the vertical coordinates for the top edges of an object.</td>
</tr>
<tr>
<td>Scale width</td>
<td>Sets the number of units for the horizontal measurement of an object’s interior</td>
</tr>
<tr>
<td>ShowInTaskbar</td>
<td>Determines whether a form or MDIForm object appears in the Windows taskbar</td>
</tr>
<tr>
<td>StartUpPosition</td>
<td>Return or sets a value specifying the position of a form when first appears</td>
</tr>
<tr>
<td>Tag</td>
<td>Stores any extra data needed for your program.</td>
</tr>
<tr>
<td>Top</td>
<td>Sets the distance between the internal top edge of an object and the top edge of its container</td>
</tr>
<tr>
<td>Visible</td>
<td>Sets a value that determines whether an object is visible or hidden.</td>
</tr>
<tr>
<td>Whatsthisbutton</td>
<td>Return whether the what’s this button appear in the title bar of a form or MDIForm</td>
</tr>
<tr>
<td>Whatsthishelp</td>
<td>Sets or return whether context-sensitive help uses what this popup is provided by windows 95 help or the main help window.</td>
</tr>
<tr>
<td>Width</td>
<td>Sets the width of an object.</td>
</tr>
<tr>
<td>Window state</td>
<td>Sets the visual state of a form window at run time.</td>
</tr>
</tbody>
</table>

Table 1: Properties and description
Add a Form:-

When we need to add more than one form in the project we can use the following two methods:

1.) By using Menu Bar:-
   In menu bar select project then select add form option and in add form dialog box click on open button, and the new form will be added in the project.

2.) By using Project Explorer:-
   In project explorer right click on from1 then select form option from the popup menu and click on the open button, and new form will be added in the project.

Fig 7: Methods of add a form
Remove A Form:-
When we need to remove the form in the project we can use the following two methods:

1.) **By using Menu Bar:-**
   In the menu bar click on project option and select REMOVE <FORM_NAME> in pop-up menu, and the form is removed from the project.

2.) **By using Project Explorer:-**
   In the Project1 window right click on Form1 (Form1) select REMOVE<FORM_NAME> in pop-up menu and the form is removed from the project.
**Saving A Form:**

When we need to save one form in the project we can use the following two methods:

1. **By using Menu Bar:**
   In the menu bar click on file option, select SAVE <FORM name> in pop-up menu then save file as dialog box is opened, give the form name and click on OK.

2. **By using Project Explorer:**
   In the Project1 window right click on Form select SAVE <form name> option from the pop-up menu, now save file as dialog box is opened give the form name and click on OK.

![Project Explorer](image1)

![Menu Bar](image2)

Fig 10: Methods of saving a form

![Window for file saving](image3)

Fig 11: Window for file saving
**The start-up form:-**

A typical application has more than single form. When we start an application the main form is loaded. By default visual basic suggests the name of the first form it created is Form1. Some time we need to change the start-up form on our requirement for example the start-up form is the addition of two number and second is multiply of two number if we want that first operation on the output console is multiply then start-up form gives this facility. If we change the name of the form, some time visual basic won’t use the name instead, when we try to start the application, visual basic will display an error message indicating that we must specify the start-up form/object.

**Following steps are used to change the start up form:-**

In the menu bar click on the project, select project properties in pop-up Menu, now project properties dialog box is opened select the general tab then select the start-up object(name of the form which we want as startup form)and click on OK.

![Fig12: Select a startup form](image)

Now when we start an application the form_lode event executed and the first form display that is set by us.
2.3 Controlling One Form within another Form

**Loading and Unloading Forms:**

To load and unload forms, use the *load and unload* statement. Once a form is loaded, it takes over the required resource, so we are unloaded a Form that’s no longer needed. When a form is unloaded, the resources it occupies are returned to the system and can be used by other forms and application.

The load method has the following syntax:-

```
LOAD FORMNAME
```

The unload method has the following syntax:-

```
UNLOAD FORMNAME
```
To unload a Form from within its own code, use the statement

Unload me

The **FORMNAME** variable is the name of the form to be loaded or unloaded. The load statement doesn’t show the form. It is only used to load a form. It is used when some forms don’t need to be displayed, they only need to be loaded in background. These forms may contain procedures needed by other application or have a special function, such as doing something in the background like “time control”.

**Showing and Hiding Forms:**

Showing Forms:-
To show a form, we use the *show* method. If the form is loaded but invisible, the show method brings the specified form on top of every other window on the Desktop. If a form isn’t loaded, the Show method loads it and then displays it.

The show method has the following syntax:-

```
FORMNAME.SHOW
```

*Show method can display form instantly it is more fast then loading a form.*

Hiding Forms:-
If our application uses many Forms, we may want to hide some of them to make room on the Desktop for other forms.

The Hide method has the following syntax:-

```
FORMNAME.HIDE
```

To Hide a Form from within its own code, use the statement

```
ME.HIDE
```

Form that are hidden are not unloaded, they remain in memory and can be displayed instantly with the SHOW method. Forms that may be opened frequently should be hidden when they are not needed.

To close all form simultaneously use

```
End
```

2.4 **Concept of MDI form**

The Multiple Document Interface (MDI) was designed to simplify the exchange of information among documents, all under the same roof. With the main application, you can maintain multiple open windows, but not multiple copies of the application. Data exchange is easier when you can view and compare many documents simultaneously.

We almost certainly use Windows applications that can open multiple documents at the same time and allow the user to switch among them with a mouse-click. Multiple MS Word windows are a typical example, although most people use it in single document mode. Each document is displayed in its own window, and all document windows have the same behavior. The main Form, or MDI Form, isn't duplicated, but it acts as a container for all the windows, and it is called the parent window. The windows in which the individual documents are displayed are called Child windows.
An MDI application must have at least two Form, the parent Form and one or more child Forms. Each of these Forms has certain properties. There can be many child forms contained within the parent Form, but there can be only one parent Form.

The parent Form may not contain any controls. While the parent Form is open in design mode, the icons on the ToolBox are not displayed, but you can't place any controls on the Form. The parent Form can, and usually has its own menu.

To create an MDI application, follow these steps:

1. Start a new project and then choose Project >>> Add MDI Form to add the parent Form.
2. Set the Form's caption to MDI Window
3. Choose Project >>> Add Form to add a SDI Form.
4. Make this Form as child of MDI Form by setting the MDI Child property of the SDI Form to True. Set the caption property to MDI Child window.

Visual Basic automatically associates this new Form with the parent Form. This child Form can't exist outside the parent Form; in other words, it can only be opened within the parent Form.

**Parent and Child Menus**

MDI Form cannot contain objects other than child Forms, but MDI Forms can have their own menus. However, because most of the operations of the application have meaning only if there is at least one child Form open, there's a peculiarity about the MDI Forms. The MDI Form usually has a menu with two commands to load a new child Form and to quit the application. The child Form can have any number of commands in its menu, according to the application. When the child Form is loaded, the child Form's menu replaces the original menu on the MDI Form.
Following example illustrates the above explanation.

* Open a new Project and name the Form as Menu.frm and save the Project as Menu.vbp

* Design a menu that has the following structure.

```vbnet
<> MDIMenu Menu caption
  • MDIOpen : opens a new child Form
  • MDIExit : terminates the application
```

* Then design the following menu for the child Form

```vbnet
<> ChildMenu Menu caption
  • Child Open : opens a new child Form
  • Child Save : saves the document in the active child Form
  • Child Close : Closes the active child Form
```

At design time double click on MDI Open and add the following code in the click event of the open menu.

`Form1.Show`

And double click on MDI Exit and add the following code in the click event

`End`

Double click on Child Close and enter the following code in the click event

`Unload Me`

Before run the application, in the project properties set MDI Form as the start-up Form. Save and run the application. Following output will be displayed.

![Fig 14: Output of MDI form](image)
And as soon as you click MDI Open you can notice that the main menu of the MDI Form is replaced with the Menu of the Child Form. The reason for this behavior should be obvious. The operation available through the MDI Form is quite different from the operations of the child window. Moreover, each child Form shouldn't have its own menu.

2.5 LET US SUM UP
In the above sections we have learned about many facts of the Form and controls depending upon the properties of Form, you may set the methods which you use to display, add, remove, show, hide, save, load and unload the Form. We also have overview of MDI form and controlling child form through parent form.

CHECK YOUR PROGRESS 2
1. What difference is between unload & hide method of a form?

2. Write a program in VB to hide and show method?

3. How will you load or activate a form?

4. What do you mean by MDI? What is an MDI form in VB?

5. Write steps to create an MDI form?

6. Why there is need to change the start-up form.
2.6 CHECK PROGRESS: The Key

CHECK YOUR PROGRESS 1

Ans 1 In Visual Basic, the basic building block of an application is a form, which is simply a window. The VB IDE can insert forms into your project, and then you can resize the forms as well as change other properties of the form. A form is distinguished from a control in that only forms can exist as standalone objects. When controls are used, they must be placed in a form.

Ans 2. There are mostly two method to open a code window
Method 1 By using project explorer, In progress explorer when we click on the view code icon then code window will open.
Method 2 When we double click on the user interface control (like command button, form etc) visual basic opens the code window.

Ans 3 The basic methods of a form is as follows:-
To show a particular form we use
Formname.show form1.show
To hide a particular form
Formname.hide
form1.hide

To close a single form use
Unload me

To load a particular form
Formname.load
form1.load

To unload a particular form
Unload formname
unload form1

To close all form at simultaneously use
End

**Ans 4** The form layout window is in the left right corner of the VB IDE. It determine the initial position of the form in our application. It also display the sequence of form that appear on run time.

**Ans 5.** The Basic properties of a form are given below:-

<table>
<thead>
<tr>
<th>Property</th>
<th>Changes...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Form Name</td>
</tr>
<tr>
<td>Caption</td>
<td>Title Bar Text</td>
</tr>
<tr>
<td>BackColor</td>
<td>Background Colour</td>
</tr>
<tr>
<td>ForeColor</td>
<td>Colour of any lines drawn during run time</td>
</tr>
<tr>
<td>BorderStyle</td>
<td>Back style - Opaque or Transparant</td>
</tr>
<tr>
<td>MaxButton</td>
<td>Whether to display a maximize button</td>
</tr>
<tr>
<td>MinButton</td>
<td>Whether to display a minimize button</td>
</tr>
<tr>
<td>MDIChild</td>
<td>Behaves as a MDI Child (window inside another)</td>
</tr>
<tr>
<td>Moveable</td>
<td>Whether the form can be moved</td>
</tr>
<tr>
<td>ShowInTaskbar</td>
<td>Whether the form appears in the Win 95 taskbar</td>
</tr>
</tbody>
</table>

**CHECK YOUR PROGRESS 2**

**Ans 1** Hiding a form remove it from the screen but the form still remain in memory and uses system resources. Unload a form means to remove the form from the memory. If we are sure that we will not need a from again then it is best to use unload option than to hide it.

**Ans 2.** A form can be shown as follows:-

```vbnet
Private Sub cmdshow_Click()
    Me.Show
End Sub
```
End Sub
and can be hidden is as follows:-

Private Sub cmdHide_Click()

Me.Hide

End Sub

Ans 3 To load a form we use
Load formname
Example: Load form1
To activate a form we user
Form name.activate
Example: Form1.activate

Ans 4. The multiple Document Interface (MDI) has parent (main window) and the child form (each document window). An MDI application must have at least two Form, the parent Form and one or more child Forms. Each of these Forms has certain properties. There can be many child forms contained within the parent Form, but there can be only one parent Form.

Ans 5. To create an MDI application, follow these steps:
1. Start a new project and then choose Project >>> Add MDI Form to add the parent Form.

2. Set the Form's caption to MDI Window

3. Choose Project >>> Add Form to add a SDI Form.

4. Make this Form as child of MDI Form by setting the MDI Child property of the SDI Form to True. Set the caption property to MDI Child window.

Visual Basic automatically associates this new Form with the parent Form. This child Form can't exist outside the parent Form; in other words, it can only be opened within the parent Form.
Ans 6 In multiple form we work with more than one form & when we execute the program the first form display, but we want other form (except form 1) to display then we change the startup form. The form which is selected as a startup form will display first when we execute the program.

2.7 REFERENCES
1. Beginning Visual Basic 6 By Peter Wright, Published by Wrox Press
2. Microsoft Visual Basic 6.0 Deluxe Learning Edition By Michael Halvorson, Published by MS Press
3. Teach Yourself Visual Basic 6 in 21 Days: Complete Training Kit by Greg Perry, Published by Sams
4. Visual Basic 6 from the Ground Up By Gary Cornell, Published by McGraw Hill
Block 3 : Object Oriented & Event Driven Programming

Unit III: Menus

3.0 Introduction

Menu bar is the standard feature of most windows applications. Menus are required for easy navigation and control of an application. In most of the windows applications, we can see menus like File, Edit, View, Tools, Help and more. Each item on the main menu bar also provides a list of options or in the form of a pull-down menu. You need not include all menu items as part of a full fledge Windows application such as Microsoft Word when you create a Visual Basic 6 program. What you need is to include those menu items that can improve the ease of using your program by the user.

In all windows applications, there are some standards followed for menus.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caption</td>
<td>Use one or two short specific words.</td>
</tr>
<tr>
<td>Organization</td>
<td>Menu items should be grouped logically by function and allow for a minimal number of levels to access each feature.</td>
</tr>
<tr>
<td>Access keys</td>
<td>Each menu item should be assigned an access key (the underlined letter in a menu or menu selection) to allow for keyboard access to the menu choices. The key should be unique in each section of the menu and is normally the first letter of the caption.</td>
</tr>
<tr>
<td>Shortcut keys</td>
<td>Any menu features that are frequently used or need to be available from any part of the program should be assigned a shortcut key. Each shortcut key can be assigned to only one menu item.</td>
</tr>
<tr>
<td>Check box</td>
<td>Menu items that simply set or clear a single program option should contain a checked feature directly in the menu.</td>
</tr>
<tr>
<td>Ellipsis</td>
<td>Each menu item that opens a dialog should be followed by an ellipsis (...).</td>
</tr>
</tbody>
</table>

Table 1
Groups of related commands found in Windows applications are called Menus. These commands depend on the application, but some common commands such as File Open and File Save are frequently found in applications. Menus are intrinsic controls, but on the other hand, menus behave differently from other controls. For example, you don't drop menu items on a form from the Toolbox; rather, you design them in the Menu Editor window, as you can see in the figure below. You invoke this tool from the Menu Editor Button on the standard toolbar or by using keyboard keys.

3.1 Objective
The Objective of this unit is to explain what is meant by Menus and how can we create menus in Visual Basic using the menu editor. We would also like that students know about what role popup menus play and the easy way to create and use popup menus. Finally we would like the student to understand various ways to create run time menus.

3.2 Visual Basic Menu Editor Dialog box
Visual Basic provides an easy way to create menus with the Menu Editor dialog. This dialog box is modal in nature. The Menu Editor command is grayed unless the form is visible. You can also display the Menu Editor window by right clicking on the Form and selecting Menu Editor option.

Basically, each menu item has following properties.

- Caption property (possibly with an embedded & character to create an access key)
- Name.

Each item also exposes three Boolean properties,

- Enabled,
- Visible, and
- Checked, which you can set both at design time and at run time.
At design time, you can assign a shortcut key to the menu item so that end users don’t have to go through the menu system each time they want to execute a frequent command.

Building a menu is a simple job. Basic steps involve:

- You enter the item’s Caption and Name
- Set other properties (or accept the default values for those properties)
- Press Enter to move to the next item.

To create a submenu, you press the Right Arrow button (or the Alt+R hot key). When you want to return to work on top-level menus—those items that appear in the menu bar when the application runs—you click the Left Arrow button (or press Alt+L). You can move items up and down in the hierarchy by clicking the corresponding buttons or the hot keys Alt+U and Alt+B, respectively.

You can create up to five levels of submenus (six including the menu bar). You can insert a separator bar using the hyphen (-) character for the Caption property. But even these separator items must be assigned a unique value for the Name property, which is a real nuisance. If you forget to enter a menu item’s Name, the Menu Editor complains when you decide to close it. The proposed convention is that all menu names begin with the three letters mnu.
Adding menu bar is relatively easy to accomplish in Visual Basic. There are two ways to add menus to your application.

Use the Visual Basic's Application Wizard

Use the menu editor.

### 3.3 Creating menu using Visual Basic's Application Wizard

To build a menu for use with your VB program, you use the Menu Editor, which appears as an icon in the toolbar of the VB IDE. It is the circled item in the screen shot below:

![Menu Editor in application Wizard](image)

**Fig 3 Menu Editor in application Wizard**

Alternatively, you can invoke the Menu Editor from the Tools menu item as shown below:

![Menu Editor from tool menu](image)

**Fig 4 Menu Editor from tool menu**

Let us understand the process of using menu editor to create menus in Visual Basic by taking an example.

**Task:** Develop File menu which will have following sub menus with it.

New

Open

Save
To build the menu described above, perform the following steps.

1. Create a new VB project
2. Open the Menu Editor using either of the following methods:
   a. click the Menu Editor toolbar icon
   b. select the Menu Editor option from the Tools menu
3. The Menu Editor screen appears, as shown below:
4. To invoke File menu using “Alt+F” from keyboard, we need to type &File in “Caption” field. This can be done by placing the ampersand to the left of the “F”, where “F” is an access key for the File item. As user presses Alt+F keys on keyboard, the File menu drops down, showing user the submenus.

5. Let’s type mnuFile in the “Name” box.

The Menu Editor screen should look like this:

![Fig 7 Menu Editor screen](image)

5. Click the Next button.

6. To create submenus, click the “right-arrow” button (shown circled below). An ellipsis (...) will appear below the &File, showing it as the next item in the menu list. This indicates that this item is a level-two item.

![Fig 8 Adding option in Menu](image)
Type &New for "caption", mnuNew in the "Name" box and select CTRL+N as "Shortcut". By specifying a shortcut, you allow the user to access the associated menu item by pressing that key combination.

So here, you are providing the user three ways of invoking the "New" function:

- clicking File, then clicking New on the menu;
- keying Alt+F+N (because we set up an access key for "N" by placing an ampersand to left of "N" in "New");
- keying Ctrl+N. At this point, your Menu Editor screen should look like this:

![Menu Editor Screen]

7. Click the Next button.

8. For "Caption", type &Open; for "Name", type mnuOpen, and for "Shortcut", select Ctrl+O.

9. Click the Next button.

10. To create a separator, type - (a hyphen) in "Caption", and for "Name", type mnuFileBar1. A single hyphen as the Caption for a menu item tells VB to create a separator bar at that location. The screen will look like:
11. Create all rest of menus option in same way. Once done, the screen should look like the following:

Fig 10 Adding separator in Menu option

12. Click the "right-arrow" button to create a level-two item below "Help". For "Caption", type &About; and for "Name", type mnuAbout. Your Menu Editor screen should look like this:

Fig 11 Adding all menu items
13. Menu creation is done as per the example we have taken. To finish things, click the OK button. Menu editor will close and the focus goes back to VB IDE.

14. In the VB IDE, the form will now have a menu on top as per the task we have just completed. If we click on File menu, it will drop down as per the screenshot:

Fig 12 Adding Second level Menu

Check Your Progress 1

Notes: i) Write your answers in the space given below.
ii) Compare Your Answer with those given at the end of the unit.

a) What is meant by Menu?

b) Explain the purpose of separator bars?

c) What is meant by checked items?

d) What is the difference between an access key and a shortcut key?

3.4 Writing events for menu options

Designing menu just creates a visual layout of the menus. However, clicking these will not do anything. In order to make these menu options work as intended, we have to write code. We need to write events which will execute when the menu item is selected and clicked. Click is the only event that a menu item can respond to. It can be considered as a single event procedure: the Click event procedure. The Click event procedure is the place where you write or call the code that you want to execute when the user chooses the menu item. You can access a menu control's Click event procedure by single clicking the menu item from the design time copy of the menu.

For example, if we want to close the VB form by clicking on Exit option, we need to write code for the click event of Exit menu option.

To write these events, we need to start from VB IDE again.

1. Click on the New menu item. The code window for the mnuFileNew_Click event opens, as shown below.
Each event opens with some predefined code, which is basically the start and end of the event. All code to be written should be placed between the start and the end.

For example, if we want to display a message when a user clicks the New menu option, we should do something like this:

```vbnet
MsgBox "message", vbInformation, "Menu Example"
```

In the `mnuFileNew_Click` event, place the code you want to execute when the user clicks the New menu item. Since this is just a demo, we will place a simple `MsgBox` statement in the event procedure:

```vbnet
Private Sub mnuFileNew_Click()
    MsgBox "Code for 'New' goes here.", vbInformation, "Menu Demo"
End Sub
```

2. Code similar `MsgBox` statements for the Open, Save, Save As, and Print menu items:

```vbnet
Private Sub mnuFileOpen_Click()
    MsgBox "Code for 'Open' goes here.", vbInformation, "Menu Demo"
End Sub

Private Sub mnuFileSave_Click()
    MsgBox "Code for 'Save' goes here.", vbInformation, "Menu Demo"
End Sub

Private Sub mnuFileSaveAs_Click()
    MsgBox "Code for 'Save As' goes here.", vbInformation, "Menu Demo"
End Sub
```
Private Sub mnuFilePrint_Click()
    MsgBox "Code for 'Print' goes here.", vbInformation, "Menu Demo"
End Sub

2. For the Exit menu item Click event, code the statement Unload Me.

Private Sub mnuFileExit_Click()
    Unload Me
End Sub

The program is ready to run now. Note how the code executes when you click on the various menu items. Also test the use of the access keys (e.g., Alt+F+N) and shortcut keys (e.g., Ctrl-O).

3.5 Popup menus in VB

Visual Basic also supports pop-up menus, which are context-sensitive menus that most commercial applications show when you right-click on an user interface object. In Visual Basic, you can display a pop-up menu by calling the form's PopupMenu method.

After the desired menus have been created, the next step for the programmer is to decide which objects will call which menus. The form, as well as individual controls on the form, all can have pop-up menus specified.

The standard method used to activate an object's pop-up menu is to use the MouseUp event procedure to detect when the user right-clicks the mouse. After the user right-clicks, the menu is displayed.

To determine the state of the mouse, the programmer traps the MouseUp event of the object to provide the menu. Using the event's Button parameter, the programmer verifies which mouse button the user pressed.

To determine which mouse button was activated, the procedure is passed two arguments: Button and Shift. The Button argument indicates the mouse button that was pressed. The Shift indicates whether the Ctrl and/or Alt and/or Shift was active when the mouse click event occurred.

The following code in the form's MouseUp event determines which mouse button was pressed and then further distinguishes which Shift key or Shift key combination was pressed along with the mouse button.

Sub Form_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)
    Dim blnIsAlt as Boolean
    Dim blnIsCtrl as Boolean
End Sub
Dim blnIsShift as Boolean
BlnIsAlt = Shift And vbAltMask
BlnIsCtrl = Shift And vbCtrlMask
BlnIsShift = Shift And vbShiftMask
If Button = vbLeftButton Then
    If blnIsAlt And blnIsShift Then
        … So something to react to Left Button + Alt + Shift
    End If
ElseIf Button = vbRightButton Then
    Form1.PopupMenu mnuPopUp1
End If
End Sub

The preceding code uses VB constants to determine the button-Shift key combination. The key combination of Ctrl and/or Alt and/or Shift will also be returned as a number in the Shift parameter. You can test to see whether a particular key was pressed by the user by using the AND operator to compare the Shift parameter with one of the bit mask constants vbAltKey, vbShiftKey, or vbCtrlKey.

In the MouseUp event, both Button and Shift are integer values. When programming for this event, either the integer value can be used or the VB constants that refer to the various possible Button and Shift key values.

This example shows you how to create a popup menu (sometimes called a context menu or a right-click menu).

1. Start a new VB project and place a label on the form. Name the label lblTestText. Set the Caption to Test Text.

![Form design view](image)

2. Open the Menu Editor, and create a top-level item with a Caption value of PopUpFormat and the Name mnuPopuUpFormat. Also importantly uncheck the Visible checkbox (see the circled item below). In order for a menu to be a pop-up menu, it must be invisible.
3. Create the following level-two menu items below the PopUpFormat top-level menu. (When creating these level-two items, keep the Visible box checked.)

<table>
<thead>
<tr>
<th>Caption</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bold</td>
<td>mnuBold</td>
</tr>
<tr>
<td>Italic</td>
<td>mnuItalic</td>
</tr>
<tr>
<td>Underline</td>
<td>mnuUnderline</td>
</tr>
<tr>
<td>-(hyphen)</td>
<td>mnuFormatSep</td>
</tr>
<tr>
<td>Cancel</td>
<td>mnuCancel</td>
</tr>
</tbody>
</table>

When done, your Menu Editor should look like this:

4. Click OK to save your changes.
Note: When you return to the IDE, you will NOT see this menu on the form (remember it’s a pop-up menu, and it will only be visible when invoked through code).

5. Code the lblTestText_MouseDown event as shown below. Note that the Button parameter is tested for vbRightButton as is conventional, we only want to pop up the menu if the user right-clicks on the label. If the user clicks the right mouse button, the PopupMenu statement is executed. It is this statement that makes the pop-up menu appear.

```vba
Private Sub lblTestText_MouseDown(Button As Integer, Shift As Integer, X As Single, Y As Single)
    If Button = vbRightButton Then
        PopupMenu mnuPopUpFormat, vbPopupMenuRightButton
    End If
End Sub
```

The full syntax for the PopupMenu method, from MSDN, is:

```
object.PopupMenu menuname, flags, x, y, boldcommand
```

The PopupMenu method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Optional. An object expression that evaluates to an object in the Applies To list. If object is omitted, the form with the focus is assumed to be object.</td>
</tr>
<tr>
<td>Menuname</td>
<td>Required. The name of the pop-up menu to be displayed. The specified menu must have at least one submenu.</td>
</tr>
<tr>
<td>Flags</td>
<td>Optional. A value or constant that specifies the location and behavior of a pop-up menu, described as follows:</td>
</tr>
<tr>
<td></td>
<td>Constant (location)</td>
</tr>
<tr>
<td></td>
<td>vbPopupMenuLeftAlign</td>
</tr>
<tr>
<td></td>
<td>vbPopupMenuCenterAlign</td>
</tr>
<tr>
<td></td>
<td>vbPopupMenuRightAlign</td>
</tr>
<tr>
<td></td>
<td>Constant (behavior)</td>
</tr>
<tr>
<td></td>
<td>vbPopupMenuLeftButtonDown</td>
</tr>
</tbody>
</table>
An item on the pop-up menu reacts to a mouse click when you use either the right or the left mouse button.

| vbPopupMenuRightButton | 2 | An item on the pop-up menu reacts to a mouse click when you use either the right or the left mouse button. |

Note: To specify both a "location" constant and a "behavior" constant, add the two values together. For example:

| PopupMenu MyMenu, vbPopupMenuRightAlign + vbPopupMenuRightButton |

| X | Optional. Specifies the x-coordinate where the pop-up menu is displayed. If omitted, the mouse coordinate is used. |
| Y | Optional. Specifies the y-coordinate where the pop-up menu is displayed. If omitted, the mouse coordinate is used. |
| Bold command | Optional. Specifies the name of a menu control in the pop-up menu to display its caption in bold text. If omitted, no controls in the pop-up menu appear in bold. |

6. Code the mnuBold_Click event as shown below. Note that the Checked property of the menu item is used. When set to True, this causes a checkmark to appear to the left of the menu item. The Checked property is typically used as a toggle.

**Private Sub mnuBold_Click()**

    If mnuBold.Checked Then
        lblTestText.FontBold = False
        mnuBold.Checked = False
    Else
        lblTestText.FontBold = True
        mnuBold.Checked = True
    End If

**End Sub**

7. Code the mnultalic_Click and mnuUnderline_Click events in a similar fashion as shown below.
Private Sub mnuItalic_Click()
    If mnuItalic.Checked Then
        lblTestText.FontItalic = False
        mnuItalic.Checked = False
    Else
        lblTestText.FontItalic = True
        mnuItalic.Checked = True
    End If
End Sub

Private Sub mnuUnderline_Click()
    If mnuUnderline.Checked Then
        lblTestText.FontUnderline = False
        mnuUnderline.Checked = False
    Else
        lblTestText.FontUnderline = True
        mnuUnderline.Checked = True
    End If
End Sub

8. Run the program and check out the various options you have coded.

Fig 19 Checked option for menu
9. Save the program and exit VB.

Check Your Progress 2

Notes: i) Write your answers in the space given below.
     ii) Compare Your Answer with those given at the end of the unit.

a) What is a popup menu?

b) What is the need to determine about mouse state while creating popup menus?

3.6 Creating Dynamic Menu

Since the menus cannot be created during runtime, complete dynamic control over the menu is not available. However since VB6 allows Array of menu items using index property, this feature can be exploited to create some dynamism.

Runtime menus are created dynamically by the application, as they are required. The runtime menu may list user-selected information, recently used files, or a list of Internet Web sites, to name three examples.

To create runtime menus dynamically, you must use a menu control array. At runtime the Load statement creates the new menu items, and the Unload statement removes them.

The three steps to creating a runtime menu item are as follows:

1. Create a design time menu that will act as the template, as shown in Figure. Set the Index property of the template item to create a menu control array. This array will allow new elements to be created at runtime. These elements can then be controlled like a regular menu item.
Use the Load statement at runtime. The Load statement accepts the name of the object to be loaded. The following is sample syntax: Load mnuFileItem(1)

2. The name of the menu item is based on the design time item that had the Index property set. To create a new menu, just use the menu name and a unique number that has not been previously used in the array.

3. After the control has been loaded, use the menu name and the index value to refer to the new menu. The menu item can then have all the normal properties of menus set at runtime.

Dynamically created menu items appear directly under the preceding index item. This must be taken into consideration when incorporating menu items below the array. Menu items below the array will function as expected; however, as the new elements are added to the collection, regular menu items appear lower on the menu. Compare the positions of the Exit menu item on the File menu before and after dynamic menu items have been added, as shown in following Figures.
When a menu item has been created at runtime, it is part of a control array. When code is to be associated with the runtime-generated menu, just use the design time menu item that was the first index number in the array.

The template menu item will have an extra argument in the Click event. The Index argument provides the number used to refer to that control. The following sample code demonstrates one way to code for the dynamic menus:

```vba
Sub mnuFileItem_Click(Index as Integer)
    Select Case Index
    Case 0
        MsgBox "You clicked the first menu item!"
    Case 1
        MsgBox "You clicked the first dynamically created menu item!"
    Case 2
        MsgBox "You clicked the second dynamically created menu item!"
    End Select
End Sub
```

Of course, code such as the above would be appropriate only when you knew ahead of time exactly which items you would be adding to your menu at runtime and how many maximum items there would be.
Removing Runtime Menu Items

You can use two different methods to remove the runtime menus. The first is to hide the newly created item; the second is to unload it. When hiding a menu item, the user interface will no longer display the item; however, program code can still use the menu and control the properties.

`mnuFileItem(1).Visible = False`

If a runtime menu item is unloaded, that control and the associated properties will be removed from memory. If required again, they will have to be loaded.

`Unload mnuFileItem(1)`

Only runtime control names and elements can be passed to the Unload statement. If a design time menu item is passed to Unload, an application error will occur because you can't unload controls created at design time.

Check Your Progress 3:

a) Using Visual Basic Menu editor, create the following form and write code to make the menu options work.

3.7 Let Us Sum Up
Menus are the key to navigation inside any website or a desktop application. VB 6.0 allows us to create user defined menus using Menu editor and run time menus. It also gives us feature for creating popup menus which get invoked by right click of mouse. All the menus are associated with some action codes which are written in IDE in Mouse click events.

3.8 Check your progress: the Key

Check your progress 1

a). What is meant by Menu?
Menus are required for easy navigation and control of an application. In most of the windows applications, we can see menus like File, Edit, View, Tools, Help and more. Each item on the main menu bar also provides a list of options or in the form of a pull-down menu.

b). Explain the purpose of separator bars?
A separator bar is displayed as a horizontal line between items on a menu. On a menu with many items, you can use a separator bar to divide items into logical groups.

c). What is meant by checked items?
A checked menu item maintains state of being either set or unset. When a Checked item is set, a check mark appears and when they are unset the check mark disappears.

d). What is the difference between an access key and a shortcut key?
As you create a menu system, you should consider ease of access to the system, and you must assign tasks to the system. You must give menus and menu items tasks to perform, such as displaying forms, toolbars, and other menu systems. You should define access keys to permit entry to the menu system. You can also add keyboard shortcuts and enable or disable menu items for more control.

Well-designed menus have access keys for quick keyboard access to the menu functionality. The access key is represented by the underlined letter in the menu title or menu item.

In addition to assigning access keys, you can specify keyboard shortcuts for menus or menu items. As with access keys, keyboard shortcuts let you choose a menu or menu item by holding down one key while pressing another. The difference between access keys and keyboard shortcuts is that you can use a keyboard shortcut to choose a menu item without first displaying its menu.

Check your progress 2

a). What is a popup menu?
A context menu (also called contextual, shortcut, and popup or pop-up menu) is a menu in a graphical user interface (GUI) that appears upon user interaction, such as a right mouse click or middle click mouse operation. A context menu offers a limited set of choices that are available in the current
state, or context, of the operating system or application. Usually the available choices are actions related to the selected object.

b). What is the need to determine about mouse state while creating popup menus?
The standard method used to activate an object's pop-up menu is to use the MouseUp event procedure to detect when the user right-clicks the mouse. After the user right-clicks, the menu is displayed. To determine the state of the mouse, the programmer traps the MouseUp event of the object to provide the menu.

3.9 References
1. Murach's Visual Basic 6: Beginner to Pro
2. Visual Basic 6.0 by Techmedia
CS – 07 Programming in VB

Block 3 : Object Oriented & Event Driven Programming

Unit IV: Dialog boxes

4.0 Introduction
4.1 Objective
4.2 About Dialog boxes
4.3 Flags and Cancel Error
4.4 Common Dialog Demo Program
4.5 Let Us Sum Up
4.6 Check your progress: the Key
4.7 References

4.0 Introduction

The Common Dialog control provides a standard set of dialog boxes for operations such as opening, saving, and printing files, as well as selecting colors and fonts and displaying help. Any six of the different dialog boxes can be displayed with just one Common Dialog control. A particular dialog box is displayed by using one of the six "Show..." methods of the Common Dialog control:

- ShowOpen
- ShowSave
- ShowPrinter
- ShowColor
- ShowFont
- ShowHelp

The Common Dialog control not an intrinsic control; rather, it is an "Active X" control that must be added to the toolbox via the Components dialog box, as shown below. This dialog box is accessed via the Project menu, Components item. Once you check "Microsoft Common Dialog Control 6.0" and click OK, the control is added to your toolbox. Then you can double-click it to make it appear on your form, as you would with any other control. The Common Dialog control is not visible at run-time.

4.1 Objective:

The Objective of this chapter is to explain what is meant by Dialog boxes and how can we use them in Visual Basic. We would like students to know about what role dialog boxes play and what is the easy way to create and use them. Finally we would like the student to understand various types of dialog boxes like Open, Save, color, Font, Printer and Help, available to them to use in their applications.
4.2 About Dialog boxes

Dialog box in Visual Basic is presented as a control. However, it is not visible by default. To add this control to the VB controls toolbox, you need to open the Components Dialog box, and check the Microsoft Common Dialog Control (SP3). Clicking will add this control to your toolbox.

![The Components Dialog Box and Toolbox](image)

Certain functionality for the dialog boxes is provided automatically by VB and Windows, but other functionality must be coded. For example, with the Open and Save dialog boxes, the functionality to navigate to different drives and directories is built in, but the functionality to actually save or open a file must be coded in your program.

Let us take a look at the two properties of the Common Dialog control

4.3 Flags and CancelError.

a) The Flags Property
The appearance and behavior of the dialog boxes can be modified to some degree with the Flags property of the Common Dialog control. Wherever possible, you should use the Flags property to control the appearance of the dialog box.

(1) Shown below is a screen shot of the Open Common Dialog Box. Note that the item “Open as read only” is circled. You can use the Flags property to suppress this item. This means that you should only show it the user decide whether or not to open the file as read-only.

![Open Common dialog box](image)

(2) Shown below is a screen shot of the Print Common Dialog Box. Note that the circled items in the “Page Range” area (for printing a selection or a specific page range) are grayed out. These items were disabled via Flags property settings. You should only enable such options if your program contains the logic to act on them.
(3) Shown below is a screen shot of the Font Common Dialog Box. The Flags property can control whether or not the "Effects" area (circled) is displayed.

b) The CancelError Property
In order to test whether or not the user clicked the Cancel button on a Common Dialog box, you must set up special error-handling logic. If you don’t, VB will treat the clicking of the Cancel button as an unhandled error and will thus cause your program to end abruptly (this will not exactly endear you to your users).

You can test to see whether or not the user clicked the Cancel button by setting the CancelError property to True – this causes an error routine you have written (using the "On Error Goto label technique) to be executed when the user clicks Cancel.

The basic structure of a VB procedure that displays a Common Dialog box should be as follows:

```vbnet
Private Sub WHATEVER()
    /* DECLARE LOCAL VARIABLES */
    ON ERROR GOTO CANCELERROR_ROUTINE
    SET CancelError PROPERTY TO TRUE (e.g., CommonDialog1.CancelError = True)
    SET Flags AND OTHER PROPERTIES TO APPROPRIATE VALUES
    SHOW THE COMMON DIALOG (e.g., CommonDialog1.ShowSave)

    ' When you show the common dialog box, processing is suspending until the user performs an action. If they click the Cancel button, VB will jump to the error-handling routine you set up (i.e., CANCELERROR_ROUTINE).

    ' Assuming the user did not click Cancel, continue processing below ...

    ' Now that the dialog box is done being shown, we can put a "normal" error handling routine in place, if desired ...

    On Error GoTo NORMAL_ERROR_ROUTINE

    NORMAL PROCESSING HERE

    ' Exit here so that you don't fall through to the error-handling logic
    Exit Sub

    NORMAL_ERROR_ROUTINE:
    MsgBox "Err # " & Err.Number & " - " & Err.Description, vbCritical, "Error"
    Exit Sub

    CANCELERROR_ROUTINE:
    ' Do nothing, no problem if user clicks Cancel button ...
```
Check Your Progress 1

Notes: i) Write your answers in the space given below.
   ii) Compare Your Answer with those given at the end of the unit.

a) What is meant by Dialog box?

b) How can you add a dialog boxes control in toolbar?

c) What is meant by Flags?

d) What is the use of CancelError property?

4.4 COMMON DIALOG DEMO PROGRAM

The Common Dialog demo program shows how to use each of the six types of dialog boxes. The design-time form is shown below, with the names and property settings for each control on the form, as well as for the form itself, shown in callouts.
The coding for the demo program is provided below, with explanations.

**The General Declarations Section**

In the General Declarations section, a form-level String variable, `mstrLastDir`, is declared. This variable is used to store the directory of the last file accessed by an Open or Save operation, so it can be subsequently used as the default directory for the next Open or Save operation.

```
Option Explicit

Private mstrLastDir As String
```

**The Form_Load Event**

In the Form_Load event, the variable `mstrLastDir` is initialized to path where the program is running; this will serve as the default directory the first time that an Open or Save operation is performed.

```
Private Sub Form_Load()
    mstrLastDir = App.Path
End Sub
```
The cmdOpen_Click Event

This event fires when the user clicks the Open button. The procedure follows the structure described above in the section discussed the CancelError property. Local variables are declared, an On Error statement is set up as part of the Cancel error handler, and then various properties of the Common Dialog box (dlgDemo) are set.

As discussed above, setting the CancelError property to True enables us to handle the run-time error that will occur if the user clicks the dialog's Cancel button.

The InitDir property tells the Common Dialog control where to start its navigation of the file system. As discussed above, we are using the form-level variable mstrLastDir to value this property.

The Flags property, as discussed earlier, is used to modify the appearance and/or behavior of the dialog boxes. The Flags property is set by assigning one or more predefined VB constants (all beginning with the letters "cdl") to it. If more than one flag is to be assigned to the Flags property, you do so by adding the constants together:

```
CommonDialog1.Flags = cdlThisFlag + cdlThatFlag + cdlTheOtherFlag
```

Multiple flags can also be assigned by joining them together with the logical Or operator:

```
CommonDialog1.Flags = cdlThisFlag Or cdlThatFlag Or cdlTheOtherFlag
```

In this case, we are assigning one flag to the Flags property, cdlOFNHideReadOnly, which tells the Common Dialog control to suppress the "Open as read only" checkbox.

The FileName property is initially set to "" so that a filename is not pre-selected when the dialog box is displayed (if we wanted to, we could initialize it with a default filename). After the Common Dialog has been shown and the user has interacted with it, we use the FileName property to determine what file the user has selected.

The Filter property is a pipe-delimited string specifying what types of files should be shown in the file list portion of the Common Dialog box. The Filter string specifies a list of file filters that are displayed in the Files of type drop-down box. The Filter property string has the following format:

```
description1 | filter1 | description2 | filter2...
```

*Description* is the string displayed in the list box — for example, "Text Files (*.txt)." *Filter* is the actual file filter — for example, "*.txt." Each *description* | *filter* set must be separated by a pipe symbol (|).

The ShowOpen method displays the Open Common Dialog box. At this point, processing will pause until the user either selects a file to open and clicks OK, or clicks Cancel. If the user clicks Cancel, the process will branch to the cmdOpen_Click Exit label and the procedure will end. If the user selects a
file and clicks OK, processing will continue with the statement following the execution of the ShowOpen method.

Assuming the user selects a file and clicks OK, the FileName property, which now contains the full path and filename of the file that the user selected, is assigned to the local variable strFileToOpen. The remaining code actually opens the file and loads into the multi-line textbox (the code uses techniques discussed in previous topics). The statement prior to Exit Sub resets the mstrLastDir variable to reflect the path of the file that the user had selected.

Private Sub cmdOpen_Click()

    Dim strBuffer As String
    Dim intDemoFileNbr As Integer
    Dim strFileToOpen As String

    On Error GoTo cmdOpen_Click_Exit

    With dlgDemo
        .CancelError = True
        .InitDir = mstrLastDir
        .Flags = cdlOFNHideReadOnly
        .FileName = ""
        .Filter = "Text Files(*.txt)|*.txt|All Files(*.*)|*.*"
        .ShowOpen
        strFileToOpen = .FileName
    End With

    On Error GoTo cmdOpen_Click_Error

    intDemoFileNbr = FreeFile
    Open strFileToOpen For Binary Access Read As #intDemoFileNbr
    strBuffer = Input(LOF(intDemoFileNbr), intDemoFileNbr)
    txtTestFile.Text = strBuffer
    Close #intDemoFileNbr

    mstrLastDir = Left$(strFileToOpen, InStrRev(strFileToOpen, "\") - 1)

    Exit Sub

    cmdOpen_Click_Error:
        MsgBox "The following error has occurred:" & vbCrLf & "Err # " & Err.Number & " - " & Err.Description, vbCritical, "Open Error"

    cmdOpen_Click_Exit:
        End Sub
Screen shot of the main form after the user has opened the “CDL_TEST.txt” File:
The cmdSave_Click Event

This event fires when the user clicks the **Save** button. In this demo application, the contents of the multi-line textbox will be saved to filename specified. The code for this event procedure is very similar to that of the cmdOpen_Click event, with the obvious differences that the **ShowSave** method of the Common Dialog control is invoked, and that we are writing out a file rather than reading one in.

We are assigning two flags to the Flags property, **cdlOFNOverwritePrompt** (which will cause the Common Dialog control to automatically prompt the user for confirmation if they attempt to save a file that already exists) and **cdlOFNPathMustExist** (which specifies that the user can enter only valid paths - if this flag is set and the user enters an invalid path, a warning message is displayed).

```vba
Private Sub cmdSave_Click()
    Dim strBuffer As String
    Dim intDemoFileNbr As Integer
    Dim strFileToSave As String

    On Error GoTo cmdSave_Click_Exit

    With dlgDemo
        .CancelError = True
        .InitDir = mstrLastDir
        .Flags = cdlOFNOverwritePrompt + cdlOFNPathMustExist
        .FileName = ""
        .Filter = "Text Files(*.txt)|*.txt|All Files(*.*)|*.*"
        .ShowSave
        strFileToSave = .FileName
    End With

    On Error GoTo cmdSave_Click_Error

    intDemoFileNbr = FreeFile
    Open strFileToSave For Binary Access Write As #intDemoFileNbr
    strBuffer = txtTestFile.Text
    Put #intDemoFileNbr, , strBuffer
    Close #intDemoFileNbr

    mstrLastDir = Left$(strFileToSave, InStrRev(strFileToSave, "\") - 1)

    Exit Sub

    cmdSave_Click_Error:
    MsgBox "The following error has occurred:" & vbNewLine & "Err # " & Err.Number & " - " & Err.Description, _
```

209
vbCritical, _
"Save Error"

cmdSave_Click_Exit:
End Sub

**The cmdPrint_Click Event**

This event fires when the user clicks the **Print** button. In this demo application, this will cause the contents of the multi-line textbox to be printed to the printer specified.

We are assigning three flags to the **Flags** property, **cdIPDHidePrintToFile** (whichsuppresses the display of the "Print to file" checkbox), **cdIPDNoPageNums** (which grays out the "Pages" option of the "Page Range" area) and **cdIPDNoSelection** (which grays out the "Selection" option of the "Page Range" area).

Note that after the **ShowPrinter** method is executed, we are saving the value of the **Copies** property, as our program must implement the logic to print multiple copies if the user has selected more than one copy on the "Number of copies" spinner of the Printer Common Dialog box. Following that, various methods and properties of the Printer object are employed to carry out the task of printing the textbox contents.

```vba
Private Sub cmdPrint_Click()
    Dim intX As Integer
    Dim intCopies As Integer

    On Error GoTo cmdPrint_Click_Exit

    With dlgDemo
        .CancelError = True
        .Flags = cdIPDHidePrintToFile + cdIPDNoPageNums _
            + cdIPDNoSelection
        .ShowPrinter
        intCopies = .Copies
    End With

    On Error GoTo cmdPrint_Click_Error

    For intX = 0 To Printer.FontCount - 1
        If Printer.Fonts(intX) Like "Courier*" Then
            Printer.FontName = Printer.Fonts(intX)
            Exit For
        End If
    Next
    Printer.FontSize = 10
    For intX = 1 To intCopies
```
If intX > 1 Then
    Printer.NewPage
End If
Printer.Print txtTestFile.Text
Next
Printer.EndDoc
Exit Sub

cmdPrint_Click_Error:
    MsgBox "The following error has occurred:" & vbNewLine _
    & "Err # " & Err.Number & " - " & Err.Description, _
    vbCritical, _
    "Print Error"

cmdPrint_Click_Exit:
    End Sub

The cmdColor_Click Event

This event fires when the user clicks the Color button. In this demo application, this will cause the BackColor property of the Label lblColor to change to the selected color. In this example, no flags are set for the Color Common Dialog box. After the ShowColor method has been executed, the selected color is retrieved from the Common Dialog control's Color property.

Private Sub cmdColor_Click()
    Dim lngColor As Long
    On Error GoTo cmdColor_Click_Exit
    With dlgDemo
        .CancelError = True
        .ShowColor
        lngColor = .Color
    End With
    On Error GoTo cmdColor_Click_Error
    lblColor.BackColor = lngColor
    Exit Sub
End Sub

cmdColor_Click_Error:
    MsgBox "The following error has occurred:" & vbNewLine _
    & "Err # " & Err.Number & " - " & Err.Description, _
    vbCritical, _
    "Color Error"
cmdColor_Click_Exit:
End Sub

Screen shot of the Color Common Dialog box:

![Color Common Dialog box](image)

*Fig 8 Selecting Color Common Dialogbox*

Screen shot of the main form after the user has selected the orange color:
The cmdFont_Click Event

This event fires when the user clicks the Font button. In this demo application, this will cause the font-related attributes selected in the Font Common Dialog box to be assigned to the corresponding properties of the Label lblFont.

We are assigning three flags to the Flags property, cdlCFBoth (which Causes the dialog box to list the available printer and screen fonts), cdlCFForceFontExist (which specifies that an error message box is displayed if the user attempts to select a font or style that doesn't exist) and cdlCFEffects (which specifies that the dialog box enables strikethrough, underline, and color effects).

Private Sub cmdFont_Click()

    Dim lngFont As Long

    On Error GoTo cmdFont_Click Exit

    With dlgDemo
        .CancelButton = True
        .Flags = cdlCFBoth + cdlCFForceFontExist + cdlCFEffects
        .ShowFont
        On Error GoTo cmdFont_Click_Error
    End With

End Sub
lblFont.FontBold = .FontBold
lblFont.FontItalic = .FontItalic
lblFont.FontName = .FontName
lblFont.FontSize = .FontSize
lblFont.FontStrikethru = .FontStrikethru
lblFont.FontUnderline = .FontUnderline
lblFont.ForeColor = .Color
lblFont.Caption = .FontName & ", " & .FontSize & " pt"
End With

Exit Sub

cmdFont_Click_Error:
    MsgBox "The following error has occurred:" & vbNewLine _
        & "Err # " & Err.Number & " - " & Err.Description, _
        vbCritical, _
        "Font Error"

End Sub

Screen shot of the main form after the user has selected the Franklin Gothic Medium font (Bold, Size 24, in Navy color):
The cmdHelp_Click Event

This event fires when the user clicks the Help button. In this event, the ShowHelp method of the Common Dialog control is invoked. Unlike the other five "Show" methods, ShowHelp does not display a dialog box per se, but rather it invokes the Windows Help Engine (Winhelp32.exe) to display a help file. ShowHelp can only display help files that are in "classic" WinHelp format (these files normally have the .hlp extension) — the ShowHelp method CANNOT display an HTML-format help file (files that normally have the extension .chm).

In order for the Help file to be displayed, the HelpCommand and HelpFile properties of the Common Dialog control must be set to appropriate values. In the demo program, the HelpCommand property is set to cdlHelpForceFile (which ensures WinHelp displays the correct help file), and the HelpFile property is set to the MS-Jet SQL Help file (JETSQNL35.HLP).

Private Sub cmdHelp_Click()
    On Error GoTo cmdHelp_Click_Error
    With dlgDemo
        .CancelError = True
    End With
End Sub
HelpCommand = cdIHelpForceFile
HelpFile = App.Path & "\JETSQL35.HLP"
ShowHelp
End With

Exit Sub

cmdHelp_Click_Error:
End Sub

---

**Check Your Progress 2**

**Notes:**

i) Write your answers in the space given below.

ii) Compare Your Answer with those given at the end of the unit.

**a)** What are the dialog boxes available in VB6.0?

-------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------

**b)** Can we open a HTML file using ShowHelp?

-------------------------------------------------------------------------------------------------------------------------------
4.5 Let Us Sum Up

The Common Dialog control provides a standard set of dialog boxes for operations such as opening, saving, and printing files, as well as selecting colors and fonts and displaying help. Any six of the different dialog boxes can be displayed with just one Common Dialog control. A particular dialog box is displayed by using one of the six "Show..." methods of the Common Dialog control. The control need to be added in the toolbar before you can use it. There are few inbuilt methods and properties available for the user while he is using the common dialog.

4.6 Check your progress: the Key

Check your progress 1

a) What is meant by Dialog box?
The Common Dialog control provides a standard set of dialog boxes for operations such as opening, saving, and printing files, as well as selecting colors and fonts and displaying help. Any six of the different dialog boxes can be displayed with just one Common Dialog control.

b) How can you add a dialog boxes control in toolbar?
The Common Dialog control not an intrinsic control; rather, it is an "Active X" control that must be added to the toolbar via the Components dialog box, as shown below. This dialog box is accessed via the Project menu, Components item. Once you check "Microsoft Common Dialog Control 6.0" and click OK, the control is added to your toolbox. Then you can double-click it to make it appear on your form, as you would with any other control. The Common Dialog control is not visible at run-time.

c) What is meant by Flags?
The appearance and behavior of the dialog boxes can be modified to some degree with the Flags property of the Common Dialog control. Wherever possible, you should use the Flags property to control the appearance of the dialog box in question such that you only present features to the user that your application supports.

d) What is the use of CancelError property?
In order to test whether or not the user clicked the Cancel button on a Common Dialog box, you must set up special error-handling logic. If you don't, VB will treat the clicking of the Cancel button as an unhandled error and will thus cause your program to end abruptly. This can be done by setting the CancelError property to True – this causes an error routine you have written (using the "On Error Goto label technique) to be executed when the user clicks Cancel.
Check your progress 2
1. What are the dialog boxes available in VB6.0?
There are 6 dialog boxes available.
   - Open
   - Help
   - Font
   - Color
   - Help
   - Printer
2. Can we open a HTML file using ShowHelp?
No, we cannot do this using showhelp.

4.7 References
1. Murach's Visual Basic 6: Beginner to Pro
2. Visual Basic 6.0 by Techmedia
Block 4: BASIC CONTROLS

Unit I:
Introduction to basic controls

Unit Contains

1.0 Introduction
1.1 Objective
1.2 What are Basic Controls
1.3 Properties and Event Procedures of Basic Controls
   1.3.1 Common Properties of Controls
   1.3.2 Common Events related to controls
1.4 Basic Controls
1.5 An Application/Program in Visual Basic
1.6 Let Us Sum Up
1.7 Check Your Progress: The Key
1.8 References

1.0 Introduction

Programming in VB is a combination of visually arranging components or controls on a form, specifying attributes and action of those components and writing additional lines of code for more functionality. Since default attributes and actions are defined for controls a simple program can be created without the programmer having to write many lines of code.

Forms are created using drag and drop technique. A tool box is used to place controls on the form by double clicking on control or by drag and drop. Controls have attributes and event handlers associated with them, default values are provided when the control is created in a properties window but may be change by the programmer with the help of code. Many attribute values can be modified during the run time based on user actions or changes in the environment providing a dynamic application.

1.1 Objective

After going through this unit you will be able to design the form with the help of placing controls on it. You will also able to set design time properties of controls. Also you will able to make dynamic application by setting run time properties of controls in different event procedures associated with it.
1.2 What are Basic Controls

Basic controls are visible in Toolbox window when you load the environment. These include Label, TextBox, Command Button, ListBox, ComboBox, RadioButton and CheckBox that are used in nearly every application. There are also additional controls or we can say ActiveX controls (formerly known as OCX controls) either present in Visual Basic Package or available commercially as shareware or even as third party controls.

Basic controls have few advantages which are as follows:

1. Support for basic controls is included in the run time file that is distributed with every Visual Basic application; it means you don’t need to distribute any additional files, which greatly simplifies the installation process.

2. You can create and display basic controls faster then other external controls (controls that are not present in toolbox window) because the code for their management is already in the Visual Basic run time module and does not have to be loaded when a form references these controls for the first time.

Following is a condensed description about these controls.

**Label** control- This control serves to display static text that should not be edited by the user. It is often use to label other controls such as Textbox, ListBox and ComboBox control.

**TextBox** Control- The Textbox control is a field that contains string of characters that can be edited by the user. It can be single line (for entering simple values) or multiline (for memos and longer notes). This control is perhaps the most widely used control of any visual basic application and also exhibits rich properties and events. It is used to accept textual information from user and to display certain information back to the user.

**ListBox** control- This control contains number of items for selection and user can select one or more of them depending upon the multi select property.

**ComboBox** Control- It is said to be a combination of Textbox and ListBox because it includes several properties and events that are similar to Textbox and ListBox.

1.3 Properties and Event Procedures of basic controls

Properties are settings that control the appearance and behavior of in object or control.

```
Text1.Text="Hello"
```

For example in the above line segment of code the Text property of TextBox control, whose Name property is set to be ‘Text1’ is set to string “Hello”. In this way you can also set other properties of Text Box control or any other basic control according to object oriented concepts of Visual Basic.
As you know Visual Basic is Object Oriented Event Driven language. This means that programs interface is comprised of objects (controls, forms and so forth) and the program is coded for different actions to perform when event happens to those objects for example Click event associated with Command Button.
An event is usually initiated by the user. By anticipating the events that can occur to various objects in your program, you can write code to respond to those events appropriately. You can construct a program to respond to events by placing the code in the event procedures. An event procedure is a segment of code that is executed when a particular event occurs to particular object.

For Example Below is an event procedure defined for Click event of CommandButton, which sets Text Property of TextBox control Named Text1 to string “hello”.

```vbnet
Private Sub Command1_Click()
    Text1.Text = "hello"
End Sub
```

You can open code window by double clicking on control or form.

1.3.1 Common Properties of Controls

**Name** Property- It is very important property as it is used in a program code to identify the control, because a program is likely have many controls of the same type you can use the controls name property to identify the particular control. Every control must have a name which is represented by value of its Name property. The thing to be noticed is that every control on a particular form must have a unique name unless it is part of control array. Visual basic assigns default name to every control on a form. Basic controls support countless properties for various objects, but there is a set of properties that is shared by many objects which belong to different classes. Let us discuss these common properties.

**Left, Top, Width and Height** property- All visible objects, forms and controls expose these properties, which affect the objects position and size. These values are always relative to the objects container, which is the screen for a form and the parent form for a control. A control can also be contained in another control, which is said to be its container. In this case Top and Left properties are related to such a container control. By default these properties are measured in twips, a unit that is based on resolution independent of user interfaces, but you can switch to another units, for example pixels or inches, by setting the container’s ScaleMode property, but remember that Left, Top ,Width and Height properties for forms are always measured in twips.

**ForeColor and BackColor** property- Basic controls expose ForeColor and BackColor properties, which affect the color of the text and color of the background, respectively. The colors of the few controls like Scrollbar are specified by windows only. But you must remember that affect of these properties depends on other properties for example, setting the BackColor property of a Label control has no effect if you set the BackStyle property of
that Label to 0-Transparent.CommandButton are peculiar in that they expose BackColor property but not ForeColor property, but you will have to specify Style property to 1-Graphical to set BackColor property to any color, other than default. Visual Basic also provides symbolic constants that correspond to all the colors that appear in the system tab in the properties window at design time for example vbRed, vbWhite, and vbYellow etc. You can also use numerical, decimal or hexadecimal constant. You can also use RGB function in code to build a color which composed of its red, green and blue components like the code segment written below.

```
Text1.BackColor=RGB (0,255,255)  'red, green, blue values.
```

**Font** property-The controls that can display strings of characters expose the Font property for example CommandButton, TextBox, Label, ListBox and ComboBox etc. At design time you can set Font attribute using a common DialogBox. But you will have to do more effort for setting font at run time because Font exposes another properties also like the Name, Size, Bold, Italic, UnderLine and StrikeThrough as shown below.

```
Text1.Font.Name = "Tahoma"
Text1.Font.Size = 12
Text1.Font.Bold = True
Text1.Font.Underline = True
```

But note that fonts Size property is peculiar in that it can’t assume you to create a font other than any particular size that you want, also it does not give error in that case.

**The Caption and Text** property-The Caption property is a string of characters that appear inside a control. No basic control exposes both a Caption and Text property. Label CommandButton, CheckBox, OptionButton expose Caption property, whereas TextBox, ComboBox control exposes Text property. In general if a control exposes Text property, it also supports the SelText, SelStart and SelLength property, which returns information about the portion of text that is currently selected in control.

**The Parent and Container** property-The Parent property is a runtime only property, which returns a reference to the form that hosts the control. The Container property is also run time property, which returns reference to the container of control. Note that you cannot move a control from one form to another using the Parent property (which is read only) but you can move a control to another container by assigning different value to its Container property (which is read, write property) as given below.

' move Text1 to Picture1 container

```vba
set Text1.Container=Picture1
```

' move it back on the form surface

```vba
set Text1.Container=Form1
```
The Enable and Visible properties- By default all controls are enabled at run time. However you might want to hide them or show them in a disabled state with the help of following code

Text1.Enable = false

Disabled controls don’t react to user actions, but otherwise they are fully functional and can be manipulated through code. Visible property is used to hide the controls as shown below

Text1.Visible = false

Invisible controls are automatically disabled, so you never need to set both these properties to false.

The hWnd property-This property is available only at runtime, and returns the 32 bit integer value that windows use internally to identify a control.

The TabStop and TabIndex properties-If a control is able to receive input focus, it exposes TabStop property. Most basic controls support this property like TextBox, OptionButton, CommandButton, ComboBox and ListBox etc. If a control supports TabStop property it also supports TabIndex property, which affects the tab order sequence that is the sequence in which controls are visited when the user presses the tab key. If you set the TabStop property to false for a control than that control does not receive InputFocus while pressing tab key in a sequence.

MousePointer and MouseIcon properties-Theses properties affect the shape of mouse cursor when it moves over a control. The Mouse Icon property is used to display a common, user defined mouse cursor. In this case you must set MousePointer property to the value 99-vbCustom and then assign icon to MouseIcon property.

Tag property- This property has no special meaning. It is simply used for storing data related to control that you want to save. For example you may use it to store the initial value displayed in a control so that you can easily restore it if the user wants to undo his or her changes.

1.3.2 Common Event Procedures Related to Controls

Click and Dblclick Event

A Click event occurs when the user do left click on a control, whereas DblClick event occurs when you double-click on a control using the left mouse button. Whenever a CheckBox controls Value property changes through code, visual basic fires a Click event exactly as user had clicked on it. ListBox and ComboBox controls also fires click event whenever the ListIndex property changes. We will discuss ListIndex property while discussing ListBox control.

The Change event
The Change event is the simplest event offered by Visual Basic. Whenever the contents of a control changes, Visual Basic fires Change event. But remember that when you click on CheckBox and OptionButton control, they fires a Click event (rather than Change event). TextBox and ComboBox controls raise a change event when the user types something in the editable area of control (but note that ComboBox raises a Click event when the user selects an item from the list portion rather then type in a box.) The Change event also fires when the contents of a control are changed through code.

**The GotFocus and LostFocus event**-

GotFocus fires when a control receives the input focus and LostFocus fires when the input focus leaves it and passes to another control.

**KeyPress, KeyDown and KeyUp events**-

These events fire whenever the user presses the key while a control has input focus. The exact sequence is as follows.

1. KeyDown (the user presses the key)
2. KeyPress (visual basic translates the key into ANSI numeric code)
3. KeyUp (the user releases the key).

Note that keys that corresponds to control keys (Ctrl+x, Backspace, Enter and escape) and printable characters activate the KeyPress event and for all other keys including arrow keys, function keys, alt+x keys combinations, this event does not fire and all the KeyDown and KeyUp events are raised.

Following is example of KeyPress event

```
Private Sub Text1_KeyPress (Ascii as Integer)

MsgBox “you pressed” & Chr$(Ascii)

End sub
```

In the above event procedure we are using Visual Basic MessageBox function to show the key pressed by user. The Chr$ function is used to convert ANSI code of the key pressed into string.

The KeyDown and KeyUp events receive two parameters KeyCode and Shift. The former is the code of the pressed key and later is an integer Value that represent Ctrl, Shift and Alt keys), because the value is bitcoded you have to use AND operator to extract the relevant operation. The KeyCode parameter tells which physical key has been pressed and is therefore differ from KeyAscii parameter received by KeyPress event. For example

```
Private Sub Text1_KeyDown (KeyCode as Integer, Shift as Integer)
```
If KeyCode=VbKeyF2 and Shift=VbCtrlMask
Text1.Text = date$
End if

End sub

The above code replaces the contents of control with the date if the user presses Ctrl+F2 key.

**The MouseDown, MouseUp and MouseMove events**

The MouseDown MouseUp and MouseMove events fire when the mouse is clicked released or moved on a control respectively. All of these events receive the same set of parameters: the state of mouse buttons, the state of shift+ctrl+alt keys, and the x and y coordinates of the mouse cursor. The coordinates are always relative to the upper left corner of the control of the form.

While using these Mouse events you should notice that the x and y values are relative to the client area of the form or control and not to its external border. The coordinates (0,0) for a form object corresponds to the pixel in upper left corner below the title bar or the menu bar(if there is one). When you move the mouse cursor outside the form area, the value of coordinates might become negative or exceed the height and width of client area. Also when you press a mouse button over a form or a control and move the mouse outside its client area while keeping the mouse button pressed, the original control continues to receive mouse events and in this case the mouse is said to be captured by the control and the captured state terminates only when you release the mouse button.

MouseDown and MouseUp events are raised any time the user presses or releases the button. For example if user presses the left button and the right button (without releasing the left button), the control receives the MouseDown events and then two MouseUp events.

The Button parameter passed by MouseDown and MouseUp events represents which button has just been pressed and released respectively. The MouseMove event receives the current state of all mouse buttons. Further when the user releases the only button being pressed, Visual Basic fires a MouseUp event and then a MouseMove event, even if the mouse have not moved. The example given below shows how to trap these mouse events.

```
Private Sub Form_MouseDown (Button As Integer, _Shift As Integer, X As Single, Y As Single)
   MouseStates Button, Shift, X, Y
End Sub

Private Sub Form_MouseMove (Button As Integer, _Shift As Integer, X As Single, Y As Single)
   MouseStates Button, Shift, X, Y
End Sub
```
Private Sub Form_MouseUp (Button As Integer, _ Shift As Integer, X As Single, Y As Single)

    MouseStates Button, Shift, X, Y

End Sub

Private Sub MouseStates (Button As Integer, _Shift As Integer, X As Single, Y As Single)

    Dim mousedata As String
    mousedata = Space$(30)
    If Button And vbLeftButton Then Mid$(mousedata, 1, 4) = "Left"
    If Button And vbRightButton Then Mid$(mousedata, 5, 5) = "Right"
    If Button And vbMiddleButton Then Mid$(mousedata, 10, 6) = "Middle"
    If Shift And vbShiftMask Then Mid$(mousedata, 16, 5) = "Shift"
    If Shift And vbCtrlMask Then Mid$(mousedata, 21, 4) = "Ctrl"
    If Shift And vbAltMask Then Mid$(mousedata, 25, 3) = "Alt"
    mousedata = "(" & X & ", " & Y & ") " & mousedata
    Label1.Caption = mousedata

End Sub

Check Your Progress 1

Notes: i) Write your answers in the space given below.
    ii) Compare Your Answer with those given at the end of the unit.

a) What are advantages of basic controls?

b) What is the difference between Listbox and Combobox control?

1.4 Important Basic Controls

TextBox Control – TextBox control have many properties and events and is also among the most complex controls. The first most commonly used property is a Text property. This property represents text that is entered by the user in the TextBox that is what is displayed inside the box on the screen. If you do
not want anything in text property when the program runs, then locate the TextBox properties window and select current value Text1 and press the delete key. You can also compile this text with the help of code in the following manner in the forms load event

Private Sub Form_Load ()
    Text1.Text=" "
End Sub

Further Text property is the default property of TextBox control. The default property of control is that property of control which can be omitted in a code. For example

    Text1.Text= "hello"
    Text1 = "hello"

Above two statements are equivalent, which means visual basic understands that it is Text property to be interpreted. You can set the Alignment property of TextBox controls to left align, right align or center the contents of the control. You can prevent the user from the changing the contents of TextBox by setting its Lock property to True. If you are creating password fields, you should set the PasswordChar property to string, typically an asterisk. You can also set the ToolTipText property to text that user understands, what the TextBox control is used for. You can also make BorderLess text box control by setting there BorderStyle property to 0-none.

You can also use other properties of TextBox control like LinkMode, Linktopic, LinkItem, and LinkTimeOut to enable a control or form to communicate through DDE (dynamic data exchange) protocol with other controls or forms possibly in other application. Before the advent of OLE and COM, Dynamic Data Exchange was the preferred way for two windows programs to communicate. But now a days these properties have been maintained only for backward compatibility with applications written in previous versions of Visual Basic.

If you want to validate TextBox, you can do this with the help of CausesValidation property and event Validate. They both work together in the following way. When the input focus leaves a control, Visual Basic checks the CausesValidation property of the control which is about to receive the focus. If this property is true ,Visual basic fires the Validate event in the control that is about to lose the focus, thus gives the chance to you to validate its contents and if necessary cancel the focus shift to next(other) control and let focus to same control which is about to lose focus and validate it. For example there is a TextBox named as Txt_Name and you don’t want to let left it as empty when it is about to lost focus then you write a code to handle this situation in a following manner :

Private Sub Txt_Name_Validate (Cancel as Boolean)
    If Txt_Name.Text="" Then
        MsgBox "Please enter some text"
    End If
End Sub
Cancel= True

End If

End Sub

Here Cancel is set to True so that Visual Basic cancels the user action and takes input focus back to the Txt_Name control. But remember to check that the control which is about to receive focus (Next or other control) have CausesValidation property set to True if not (which is true by default).

Label control

As we know that Label control serves to display static text, it also provides descriptive caption through Caption property, which is default property of label control. Label control provides descriptive caption and possibly an associated hot key for other controls such as TextBox, ListBox and ComboBox, that don’t expose the Caption property. You can just place a Label control where you need it, let its Caption property to a suitable string (embedding an ampersand character in front of the hot key you want to assign. Caption is the default property of Label control. Now let us define what is hot key, when you set a Caption property of Label control with an ‘&’ as preceding character for example &width, the ‘&’ character is not echoed in the Caption property of Label. This property actually creates a hot key and associates it with the control. So whenever you want to focus control associated with it you can do this by pressing following combination of keys Alt+x key, where x is the first letter in the property or letter immediate to ampersand character in a property, here it is w.You can change Font attribute used for control through Font property. Another property is TabIndex property through which you can set the tab order of control while pressing tab key. You can use BorderStyle property of Label control if you want Label control to appear 3-D .Border Alignment property is used to align caption to the right or center it on the control.WordWrap property is set to true so that label control will expand for multiple lines instead of being truncated by the right border on the control. You can also set AutoSize property to true and let the control automatically resize itself to accommodate longer caption strings.Backstyle property is set to 0-Transparent if you want to show a character string somehow on the form but at the same time you don’t want to obscure underlying object.

This control exposes only a subset of events supported by other controls. For example Label control can never get the Input focus event, as they don’t support GotFocus or any keyboard related events. Label control support Click, Dblclick,_MOUSE_DOWN, MouseMove and MouseUp events.

ListBox control

ListBox control lets you to display the list of items which you set through List property ,but remember to press Ctrl+Enter key at the end of every string(item) to enter another string(item).You can also do this with the help of code like this:

Private sub Form_Load ()
List1.AddItem “first”
List1.AddItem “second”
......

End Sub

As you know that ListBox control contains a collection of items, you need to assign few properties to this control, for example you set Sorted property to True to create ListBox control that automatically sorts their items in alphabetical order. You can create ListBox with several columns and a horizontal scrollbar, by acting on the Columns property, but remember you can make assignments for both these properties only at design time and can’t change style of ListBox while the program is running. You set IntegralHeight property to false in properties window so that visual basic won’t enforce particular height and you are free to resize the control as you prefer. But you can modify this property at runtime. If you know at design time what items must appear in the ListBox control, you can save some code and enter items right in the properties window, in the list property mini editor by using ctrl+enter key after entering each item.

You can use AddItem method to add items when the program is executing as shown below.

Private Sub Form_Load()

List1.AddItem “first”
List1.AddItem “second”
List1.AddItem “third”
......

End Sub

If you want to add item in a given position than you should do this by specifying position of item as a second argument to the AddItem method as shown below.

List1.AddItem “zero”, 0

You can remove item from a list with a RemoveItem or Clear method.

List1.RemoveItem 0 ‘remove first item

Or

List1.Clear ‘to remove all items.

The ListIndex property returns the index of the selected item (zero based), while the Text property returns the active string in the List Box. You can also assign a value to ListIndex property to programmatically select an item or set it to -1 to deselect all items.

List1.listdir=2 ‘selects third item in the list.

The ListCount property returns the number of items in the control.
You can also enumerate the item with the help of list property as follows:

```
For i=0 to List1.Listindex-1

Print “item” & i & “=” & List1.List (i)

Next
```

ListBox exposes Click event which occurs when ever a new element has been selected (with the mouse or keyboard or programmatically)

For example

```
Private Sub List1_Click ()

Debug. Print “user selected item #” & List1.Listindex

End Sub
```

You can also use Scroll event in conjunction with TopIndex property, which sets or returns the index of the first visible item in the list area if you need to synchronize a ListBox control with the another ListBox control, you can do this with the help of Scroll event and Top Index property like this

```
Private Sub List1_Scroll ()

ListBox2.TopIndex = ListBox1.TopIndex

End Sub
```

```
Private Sub List2_Scroll ()

ListBox1.TopIndex = ListBox2.TopIndex

End Sub
```

You can associate a 32 bit integer value with each new item loaded in the ListBox control with the ItemData property as shown below:

```
List1.ItemData (listbox.listcount-1) = CustId
```

You can make ListBox more flexible so that user can select multiple items at the same time using shift and control keys. To enable this feature you can assign MultiSelect property to the values 1-simple,2-extended, but remember that it is design time property.

**ComboBox control**

ComboBox controls are very similar to ListBox control. You can create ComboBox control that automatically sorts the items using the Sorted property. Most runtime methods of ListBox like AddItem, RemoveItem, Clear and properties like ListCount, ListIndex, List, ItemData, TopIndex and events like Click, DblClick, Scroll are treated just like as ListBox control. ComboBox does not support multiple Columns and multiple selections, so you don’t have to deal with Column, MultiSelect, Select and SelCount properties.
ComboBox supports three types of styles
1 DropDown Combo (Style 0)
2 Simple Combo (Style 1)
3 DropDownList (Style 2)
When you set to Style=0 Dropdown Combo, what you get is classic Combo, you can enter a value in the edit area or select one from the dropdown list. The setting Style=1 Simple Combo is similar, but list area is always visible. The Style=2 dropdown list suppresses the edit area and gives you only the dropdown list to choose from.
ComboBox combines the feature of Textbox and ListBox as the ComboBox allows the user to select an item either by typing text into the ComboBox or by selecting it from list.
A ComboBox is appropriate when there is also a list of suggested choices, and a list box is appropriate when you want to limit input to what is on the list.

**CheckBox**

CheckBox controls are useful when you want to offer user a yes or no, true or false choice. Any time you click on this control, it toggles between the yes state and the no state. The control can also be grayed when the state of the checkbox is unavailable, but you must change the state through code.
It provides Caption property to display the descriptive strings. If you want to display control in a checked state you set its value to 1-checked right in the properties window or you can set to grayed state with 2-grayed.
The most important event for CheckBox control is Click event, which fires when either user or the code changes the state of the control. You can query the controls Value property that whether it is check or not like this:

```vba
If Check1.value Then
    MsgBox "check 1 is opted"
End if
```
The Value property is the default property of CheckBox control.

**RadioButton**

Option button controls also known as RadioButton controls. You always use OptionButton control in a group of two or more because their purpose is to offer a number of mutually exclusive choices .You can set an OptionButton controls caption property to a meaningful string. If the control is the one in its selected state, you can set its value property to true.
Check Your Progress 2
Notes: i) Write your answers in the space given below.
ii) Compare Your Answer with those given at the end of the unit.

a) What are GotFocus and LostFocus Events of Controls?

b) Write the Default Properties of following Controls.

   i. Textbox     ii. Label
   iii. ListBox   iv. ComboBox

c) What are Hot-keys?

1.5 An Application/Program in Visual Basic

Following is an application program in Visual Basic. The aim of this Program is to display the entries entered by the user in the Messagebox.

In this program as shown in Fig 1.5.1 we take four Label controls and set their name property as Lbl_Name, Lbl_Course, Lbl_Hobby, Lbl_Sex and Caption properties as NAME,COURSE,HOBBY and SEX respectively. We take one Textbox Control having Name Property set as Txt_Name. There is one ListBox and one ComboBox named as Lst_Course and Cmb_hobby respectively and their items are added with the help of runtime code as you see later. There are two Radio buttons having Caption Property set as MALE and FEMALE. There are two CheckBox controls who’s Caption Properties are set as MARRIED and NRI respectively. Then there are two Command Buttons whose name property are termed as Cmd_Submit and Cmd_Reset and Caption properties as SUBMIT and RESET.

You can align these controls with the help of Format menu given in Visual Basic IDE. Goto Format menu and then select Align submenu there are six commands that are left, center, right, top, middle and bottom. With the help of these commands you can align all controls on the form as per your requirement. Similarly there is a submenu named ‘Make same size’ in which width, height, and both commands are given so that you can make size of controls uniformly through out. You can select multiple controls by hold down a control key at design time. In the format menu there are two submenus named Horizontal spacing and Vertical spacing, you can also take help of these menus. Run time code of above program is given below next to figure.
Fig 1.5.1 Form in design View

Application Program code

Dim sex As String  'Variables for storing sex,
Dim married, nri As String 'marital and residential status
Private Sub Cmd_Submit_Click ()
If Opt_Male.Value Then  'Option Button value, which decides value of `sex` variable
    sex = "MALE"
End If

If Opt_Female.Value Then
    sex = "FEMALE"
End If

If Chk_Married.Value Then  'Checkbox value, which decides value of `married` variable
    married = "Married"
Else
    married = "Unmarried"
End If
End If
If ChK_Nri.Value Then
`nri` variable

'Checkbox value, which decides the value of

nri = "Nri"
Else
nri = "Non NRI"
End If
If Txt_Name.Text = "" Then
‗Validation code for Textbox, Listbox and
ComboBox
MsgBox "You didn‘t entered name"
ElseIf Cmb_Hobby.Text = "SELECT" Then
MsgBox "You didn‘t select hobby"
ElseIf Lst_Course.ListIndex = -1 Then
MsgBox "You have not selected a course"
Else
'Messagebox Function For displaying Entered values in a Dialog box
MsgBox "Your Name is " & Txt_Name.Text & _
" ,Your Sex is " & sex & " ,Your Hobby is " _
& Cmb_Hobby.Text & " ,Your Course is " & Lst_Course.Text & _
" and You are " & married & " " & nri
End If
End Sub
Private Sub Form_Load()
Lst_Course.AddItem "MCA", 0 'Additem Function adding items
Lst_Course.AddItem "BCA", 1 'in listbox and Combobox
Lst_Course.AddItem "BE", 2
Lst_Course.AddItem "BCom", 3
Cmb_Hobby.AddItem "Cricket", 0
Cmb_Hobby.AddItem "Music", 1
Cmb_Hobby.AddItem "Gliding", 2

234


Cmb_Hobby.AddItem "Sailing", 3
End Sub

'Validating textbox so that only non-numeric values are entered
Private Sub Txt_Name_KeyPress (KeyAscii As Integer)
If KeyAscii >= Asc ("0") And KeyAscii <= Asc ("9") Then
    KeyAscii = 0
End If
End Sub

Private Sub Cmd_Reset_Click () 'Resetting values in Form
Txt_Name.Text = ""
Lst_Course.ListIndex = -1
Cmb_Hobby.Text = "SELECT"
Chk_Married.Value = False
ChK_Nri.Value = False
End Sub
Fig 1.5.2 Form Execution

Fig 1.5.3 Message dialog Box
Check Your Progress 3

Notes: i) Write your answers in the space given below.
   ii) Compare Your Answer with those given at the end of the unit.

a) Write a code for validating textbox control so that only numeric entries are entered in a Textbox?

b) Write a code for enumerating all the items in a listbox?

---

1.6 LET US SUM UP

After going through this unit, you can able to make dynamic Visual Basic application. Let us recall what we have discussed so far.

1. Basic controls are those controls available in toolbox window of visual basic IDE. These controls are used in almost every application. These controls also display faster because the code for their management is already in visual basic runtime.

2. We can place controls on a form by selecting the control type, then click on the form and drag the mouse cursor until the control has the size and the shape you want.

3. We can set properties of controls in the properties window of Visual Basic IDE at design time or want to set these properties with the help of writing code in a code window to be executed on runtime.

4. We can write event procedures related to different controls provided by visual basic environment.

1.7 Check Your Progress: The Key
Check Your Progress 1

a) Basic controls are those controls available in toolbox window of Visual Basic IDE. These controls are used in almost every application. These controls also display faster because the code for their management is already in visual basic runtime

b) A ComboBox is appropriate when there is list of choices other than already listed in a control. ListBox is appropriate when you want to limit input to what is on the list.

Check Your Progress 2

a) GotFocus fires when control receives input focus and LostFocus fires when the input focus leaves it.

b) i Text   ii. Caption iii. Text   iv. Text

c) Hot keys are also short cut keys so that user can quickly move focus to the control.

Check Your Progress 3

a) Private Sub Txt_Name_KeyPress (KeyAscii As Integer)
   If KeyAscii < Asc ("0") And KeyAscii > Asc ("9") Then
       KeyAscii = 0
   End If
b)

For i=0 to List1.ListCount - 1
   Print "item " & i & "=" & List1.List (i)
Next

1.8 References

1. Steven Holzner, Visual Basic 6 Black Book

2. Mastering VB 6.0 (BPB)

3. Peter Norton, Visual Basic 6

4. Gary Cornell, Visual Basic from ground up (TMH)

5. VB 6 the Complete Reference.
Block 4: BASIC CONTROLS

Unit II: EXAMPLES OF BASIC CONTROLS

Unit Contains

2.0 Introduction

2.1 Objective

2.2 ScrollBar Control

2.3 Example using Option Button, Check Boxes and Scroll Bars

2.4 Timer Control

2.5 Running Lights Application

2.6 Creating a Flying Message Application

2.7 Image Control

2.8 Let Us Sum Up

2.9 Check your Progress: The key

2.10 References

2.0 Introduction

Now as you are fully acquainted with basic controls, their properties and their events. You can use these controls in your application and follow the object oriented concepts of Visual Basic and write code for these controls or objects and in this way you can rapidly develop your application. There are more basic controls like ScrollBar control, Timer control, Image control which also helps to develop more dynamic and versatile application. You can also make animated applications with the help of timer control. Timer Control is useful because it provides a sequence of pulses sending regularly in an application or form which uses Timer control so that we can synchronize our events with this Timer control every time when the Timer event of Timer control fires. Image control lets you load images like JPEG and GIF so that you will be able to develop graphical applications as well as interactive applications by converting these images into compact small Buttons (as Image control supports almost all mouse events).

2.1 Objective

After going through this unit you will be acquainted with ScrollBar control, Timer control and Image control. You can learn their properties and events procedures related with these controls. You will also able to develop dynamic application using these controls by understanding the examples give in this unit. You can use the ScrollBar controls properties and event
procedures to trap the current (at that instant) value of indicator on Scrollbar control and also able to write event procedures related with it. In this unit there are two applications called Running lights application and Flying Message application. In Running lights application you will run lights (colored lights) with the help of Timer control and in the flying message application you will move message across the form (screen) so that it looks like an animated form. So you can be able to design animated applications as well.

2.2 ScrollBar Control
There are two types of ScrollBar controls HScrollBar and VScrollBar. These controls are not much used because majority of controls display their own scrollbar control if necessary.
As you already know how to place these controls on a form, that is by drag and drop or by double clicking on Control in a toolbox window. You know that you have to scroll the slider or indicator on control for scrolling but you must know that while scrolling the control either vertically or horizontally, there is a scale of values synchronized with it which varies as you move slider that is when you are scrolling the control at run time or clicking the arrows given at the end of control. The minimum and maximum values of this scale can set by min and max properties of HScrollbar and VScrollbar control. You can set min and max properties according to the requirements of desired application it is not necessary to set min property to minimum value or max property to maximum value. For instance in the example given below you are free to set min property to 255 and max property to 0 or vice-versa. There is also important property called value which is a runtime property and it returns the relative position of indicator on the scrollbar. By default the min property correspond to the left most end in case of HScrollBar control and upper end in case of VScrollBar control. For example you can set all these properties at runtime by writing following lines of code.

Private Sub Form_Load()
    VScroll1.Min = 1
    VScroll1.Max = 1000
    VScroll1.SmallChange = 1

    HScroll1.Min = 1
    HScroll1.Max = 1000
    HScroll1.SmallChange = 1
End Sub

You can test the value of Value property with the help of following piece of code after placing Label control on the form with these ScrollBar controls.

Private Sub VScroll1_Change()
    Label1.Caption = VScroll1.Value
End sub
There are two more properties called SmallChange and LargeChange. SmallChange is the variation in value you get when clicking on the ScrollBar's arrows at the ends and LargeChange is the variation you get when clicking or the ScrollBar indicator.

There are two most important events for ScrollBar controls: The Change event and Scroll event. The Change event fires when you click on the ScrollBar arrows or when you drag the indicator. The Scroll event fires when you drag the indicator. Scrollbar control can receive the input focus and also support TabIndex and TabStop property. When scrollbar control has the focus you can move indicator using the Left, Right, Up, Down, PgUp, PgDown, Home and End keys. Now you acquainted with important properties and events of scrollbar control see how to use these properties and events.

Take a PictureBox control on a form which act as a container for other controls and put some basic controls in a PictureBox, like TextBox, Labels etc. Put HScrollBar control and VScrollBar control on a form. Now you can Scroll PictureBox control in a form with the help of following code written in procedure named movepicturebox ()

```
Sub movepicturebox()
    Picture1.Move -HScroll1.value, -VScroll1.value
End Sub.
```

That is you are assigning negative value to Left and Top properties to uncover the form near the right border and to display the portion near the forms bottom border. You can call this sub procedure in the change or scroll event. You can't easily manage scrollable forms at design time so it will be better if you work with maximized form and with picture size control as large as you wish(larger or moderately large) and reset the Windowstate property to 0-Normal.

ScrollBar controls are useful for building scrolling forms and you can easily convert regular form into scrollable one. For this you need two ScrollBar controls plus a pictureBox control that we use as a container for all the controls on the form.

One more thing, you can take two Scrollbars on either side of the application for example two Vertical Scrollbars on extreme left and extreme right. Similarly you can take two Horizontal Scrollbars on Extreme top and extreme bottom. In either case you will have to write following lines of code to work each of them properly.

```
Private Sub VScroll1_Change()
    VScroll2.Value = VScroll1.Value
End Sub

Private Sub VScroll2_Change()
    VScroll1.Value = VScroll2.Value
End Sub
```

2.3 Example using option buttons checkboxes and scrollbars.
In the following application program we are changing the BackColor and ForeColor of Label control on the basis of values set in the ScrollBar's after moving indicator on different positions. So to implement this program take three VScrollBar controls and set its minimum value to 0 and max value to 255 for all the three VScrollBar controls. Also name these three VScrollBar controls as VScroll_Red, VScroll_Green and VScroll_Blue respectively. Take one Label control and name it as Lbl_Color and set its Caption property to a string “WELCOME”. Also set its alignment property to 2-center and resize the font size accordingly. Take one more Label control and set its Caption property to string “value of backcolor”. Take two OptionButton controls and named it as Opt_Decimal and Opt_HexDecimal. Take two more label controls in front of these two OptionButton controls and name it as Label_Sdecimal and Label_Hdecimal. Take three square shaped Label controls and name it as you prefer and set its caption property to letters R, G, B respectively as shown in figure 2.3.1. Finally take two CheckBoxes and name it as Chk_back and Chk_fore.

Now see how this application works. Whenever we move the arrows or indicator on three VScrollBar control's it sets the three parameters of red green and blue values of RGB function whose min and max properties are set as 0 to 255 respectively. But remember to check one or both CheckBox controls at runtime which sets the BackColor and ForeColor of Label control. If only one of these checkboxes is checked at runtime then only one of these color changing functionalities works. That is either Foreground color that is color of Text changes or the background color changes. Further you can also set the decimal and hexadecimal values of these colors after opting one of these OptionButtons. That is at run time you can opt (selected) only one radio button. When you opt for Decimal values then it shows the decimal values of selected color and when you select the hexadecimal option than it displays the hexadecimal value of the selected color. Further you can set the scrollbar control to either 255,0,0 or 0,255,0 or 0,0,255 to set Red,Green,Blue Colors uniquely.

Following is a Design time Figure and also given runtime Figure of a form. After figures is given a source code of this program. Note that in the source code we have intentionally reverses the R G B values in foreground and backcolor of RGB function so that we can differentiate the foreground and background colors at runtime.

<table>
<thead>
<tr>
<th>CONTROL PROPERTY NAME</th>
<th>CONTROL PROPERTY VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>VScrollBar #1 Name</td>
<td>VScroll_Red</td>
</tr>
<tr>
<td>VScrollBar #2 Name</td>
<td>VScroll_Green</td>
</tr>
<tr>
<td>VScrollBar #3 Name</td>
<td>VScroll_Blue</td>
</tr>
<tr>
<td>Label #1 Name</td>
<td>Lbl_Color</td>
</tr>
<tr>
<td>Label #2 Caption</td>
<td>“WELCOME”</td>
</tr>
<tr>
<td>Label #2 Alignment</td>
<td>2-Center</td>
</tr>
<tr>
<td>Label #3 Name</td>
<td>Lbl_Show</td>
</tr>
<tr>
<td>Label #3 Caption</td>
<td>“Value of Back Color”</td>
</tr>
<tr>
<td>OptionButton #1 Name</td>
<td>Opt_Decimal</td>
</tr>
<tr>
<td>OptionButton #1 Caption</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>OptionButton #2 Name</td>
<td>Opt_HexDecimal</td>
</tr>
</tbody>
</table>
Fig 2.3.1 Design time form
Source code of above application program

'flag used in Show_Color function for signifying the status 'of CheckBox controls

Dim flag As Integer

'procedure for showing Decimal and Hexadecimal values 'of color settings in three 'VScrollBar controls on the basis of option 'Selected

Sub Show_Values ()
    If Opt_Decimal.Value Then
        'RGB function for extracting the Decimal Value of colors.
        Label_HDecimal.Caption = ""
    End If

    If Opt_HexDecimal.Value Then
        'Hex$ and RGB function for extracting the Hexadecimal values of colors
    End If
End Sub
Sub Show_Color()
    If Chk_back.Value = vbChecked And Chk_fore.Value = vbUnchecked Then
        flag = 0
    End If
    If Chk_back.Value = vbUnchecked And Chk_fore.Value = vbUnchecked Then
        flag = 4
    End If
    If Chk_fore.Value = vbChecked And Chk_back.Value = vbUnchecked Then
        flag = 1
    End If
    If Chk_back.Value = vbChecked And Chk_fore.Value = vbChecked Then
        flag = 2
    End If
    If flag = 0 Then
    End If
    If flag = 1 Then
    End If
End Sub

'Procedure Show_Color for setting the foreground and background colors of label control 'named Lbl_color on the basis of Checkboxes that are checked with the help of flag 'variable
If flag = 2 Then

Lbl_Color.BackColor=RGB(VScroll_Red.Value, VScroll_Green.Value,
Scroll_Blue.Value)

Lbl_Color.ForeColor=RGB(VScroll_Blue.Value, VScroll_Green.Value,
Scroll_Red.Value)

End If

If flag = 4 Then

Exit Sub

End If

End Sub

‘Click event procedure of checkbox named CHK_back

Private Sub Chk_back_Click ()

Show_Color

End Sub

‘Click event procedure of checkbox named CHK_fore

Private Sub Chk_fore_Click ()

Show_Color

End Sub

‘Click event procedure of OptionButton named Opt_Decimal

Private Sub Opt_Decimal_Click ()

Show_Values

End Sub

‘Click event procedure of OptionButton named Opt_HexDecimal

Private Sub Opt_HexDecimal_Click ()

Show_Values

End Sub

‘Change Event procedure associated with Scrollbar named VSCRoll_Blue
Private Sub VScroll_Blue_Change()
    Show_Color
    Show_Values
End Sub

'Change Event procedure associated with Scrollbar named VSCRoll_Green

Private Sub VScroll_Green_Change()
    Show_Color
    Show_Values
End Sub

'Change Event procedure associated with Scrollbar named VSCRoll_Red

Private Sub VScroll_Red_Change()
    Show_Color
    Show_Values
End Sub
Check Your Progress 1

Notes: i) Write your answers in the space given below.
   ii) Compare Your Answer with those given at the end of the unit.

   a) Differentiate between SmallChange and LargeChange Properties of ScrollBar control?

   b) Explain about Scroll and Change event of ScrollBar controls?

   c) When a ScrollBar control has a focus, what are the keys you use to move the Indicator of ScrollBar control?

   d) How will you synchronize two HScrollBar control or two VScrollBar control on form taken simultaneously so that if one is scrolled then the scrollbar on its opposite side also changes its position?

2.4 Timer control

The Timer control lets you to provide clock with the help of which you can fire events at particular time or with regular intervals of time. This control is invisible at run time but you will have to put it on the form to use it. The most important event procedure related with this control is the timer event procedure with the help of which you can trap time intervals or pulses of clock. This
control has only two important properties called Interval and enabled. Interval stands for number of milliseconds between subsequent pulses on which timer event fires. Enabled property lets you activate or deactivate this control. You can set interval property to any appropriate property at runtime by writing it in the form load event procedure like this

Private Sub Form_Load
  Timer1.Interval=100
End sub

Following is an example of timer event procedure

Private Sub Timer1_Timer()
  .............
End sub

For example you can develop a digital clock by writing following lines of code in a Timer event procedure of timer control.

Private sub Timer1_Timer()
  Dim str as String
  Str= Time$
  lblTime.Caption=Str
End sub

To implement above simple clock application take one label control and place it any where on the form as you wish (more appropriate is to place it some corner on the form because it is a digital clock) and set its name property lblTime. Take one timer control named as Timer1. Now in the Timer event procedure of Timer control Take one string variable and assign the value return by Time$ function to it. Then assign this string to Caption property of label control named lblTime. Now see how this application works. Each time when a Timer event fires it sets the str variable to current time with the help of Timer$ function. Then we have assigned this string to caption property of Label control named lbltime which displays the current time. But remember that the time it shows is based on the time you have set in the system clock of the computer you are working with.

One more use of timer control is to display status information of control which currently has focus. For example you want to display a short string to describe the control which currently has focus. You can perform this task easily by writing few lines of code .Take one label control on a form and named it as lbl_status. Now set the Tag property of each control of a form to a string at design time as you wish. Then place a timer control on a form with an interval
property set to 100. Now write following event procedure to complete this task in a Timer event procedure like this.

Private sub Timer1_Timer()

On error resume Next

Lbl_status.caption=Activecontrol.Tag

End Sub

2.5 Running Lights Application

In this application program we are running the lights or colors of objects according to the interval property of Timer control. To implement the application take on Label control named Label1 and create a control array (copy the control and paste it on the form as many times as required) of this Label control of size 6 having indexes ranging from 0 to 5 and set its Caption property to letters A, B, C, D, E and F respectively. Now as shown in figure 2.5:1 below take one timer control and place it at any appropriate place as you wish because it is not visible at run time. Now write the code as shown below to completely implement this program. When you run this program it shows a light running clockwise that is colors are moving clockwise in a uniform manner. You can also develop traffic light application according to the logic of this program. You can set Interval property of timer control in form load event as you wish, we set it to 1000 as shown in the code below

<table>
<thead>
<tr>
<th>CONTROL PROPERTY NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label #1 Name</td>
<td>Label1</td>
</tr>
<tr>
<td>Label #1 Caption</td>
<td>“A”</td>
</tr>
<tr>
<td>Label #2 Name</td>
<td>Label1</td>
</tr>
<tr>
<td>Label #2 Caption</td>
<td>“B”</td>
</tr>
<tr>
<td>Label #3 Name</td>
<td>Label1</td>
</tr>
<tr>
<td>Label #3 Caption</td>
<td>“C”</td>
</tr>
<tr>
<td>Label #4 Name</td>
<td>Label1</td>
</tr>
<tr>
<td>Label #4 Caption</td>
<td>“D”</td>
</tr>
<tr>
<td>Label #5 Name</td>
<td>Label1</td>
</tr>
<tr>
<td>Label #5 Caption</td>
<td>“E”</td>
</tr>
<tr>
<td>Label #6 Name</td>
<td>Label1</td>
</tr>
<tr>
<td>Label #6 Caption</td>
<td>“F”</td>
</tr>
<tr>
<td>Timer#1 Name</td>
<td>Timer1</td>
</tr>
<tr>
<td>Timer #1 Enabled</td>
<td>True</td>
</tr>
<tr>
<td>Timer #1 Interval</td>
<td>1000</td>
</tr>
</tbody>
</table>

Here you have seen that all Label controls are named as Label1 because we have created a control array by creating one Label control on a form, then copy the control, and then paste it five times on the form and then set the
properties of these controls according to data you have seen above. These Controls are identified in the source code by indexing those controls like Label1 (0), Label1 (1), Label1 (2) and so on.
Source code for running lights application
Private Sub Form_Load()

‗Timer’s Interval property set to 1000
Timer1.Interval = 1000

‗Control array of Label control’s named Label1 and
‗assigning the different color constants so that each
‗label control shows different color
Label1(0).BackColor = vbMagenta
Label1(1).BackColor = vbBlue
Label1(2).BackColor = vbCyan
Label1(3).BackColor = vbGreen
Label1(4).BackColor = vbRed
Label1(5).BackColor = vbYellow
End Sub

‗Timer event fires every 1000 milliseconds
Private Sub Timer1_Timer()
Dim i as Long
Dim temp As String

‗ temp is a temporary variable for saving the color state of first label
‗control of control array
temp = Label1(Label1.Count - 1).BackColor

‗Loop which rotates the colors of control on every timer event
For i = 5 To 1 Step -1
Label1(i).BackColor = Label1(i - 1).BackColor
Next

Label1(Label1.LBound).BackColor = temp
End Sub
2.6 Creating a flying message application

In flying message application a message is flying across the screen according to settings we have configured in source code. To implement this program take one Label control named label1 and set its Caption property to a string ‘WELCOME TO FLYING MESSAGE APPLICATION’ and backcolor property to Red color. To completely implement this program see the source code below. When you run this application you see the message flying across the screen. When you maximize the form window at run time you see the complete functionality of this application. We have also take one timer control and set its interval property to 1 in form load event.
Source code of Flying message application

'variables for storing the top and left values of control holding a message
Dim t, l As Integer

Private Sub Form_Load()
    'Timer interval property set to 1
    Timer1.Interval = 1
    Form1.Width = 8000
    Form1.Height = 8000
    Label1.Width = 2000
    Label1.Height = 800
    t = 20
    l = 20
    End Sub

    'Timer event which fires every 1 millisecond and
    'changes the position of control holding message accordingly
    Private Sub Timer1_Timer()
        If Label1.Top <= 0 Then
            t = 20
        End If
        If Label1.Left <= 0 Then
            l = 20
        End If
        If Label1.Top >= Form1.Height - Label1.Height Then
            t = -20
        End If
        If Label1.Left >= Form1.Width - Label1.Width Then
            l = -20
        End If
        Label1.Top = Label1.Top + t
Label1.Left = Label1.Left + 1

End Sub

2.7 Image Control
With the help of image control you can display images. There is also one more control named PictureBox control which lets you display images but the difference is that image control is less complex than PictureBox control. Further image control load picture faster than PictureBox control and also consumes less memory and system resources. To load a picture in image control you have to use picture property at design time. If you want to load picture at runtime you have to use load picture function. Following is the example of load picture function

Image1.Picture = LoadPicture ("C:\Windows\image1.gif")

Image control does not expose AutoSize property because this control resizes to display the contained Image by default. Image control also supports Stretch property. If Stretch property is true then control resizes the image to fit in the control and if necessary distort it. Image control support all mouse events. So you can also use image control as a graphic button or in tool bar. Image Controls have several limitations. Image control does not support the graphical methods. Further they also don't support the AutoRedraw and the ClipControls Properties. They also don't work as containers. Image control is Windowless objects and is managed by Visual Basic without creating Windows object.

Check Your Progress 3

Notes: i) Write your answers in the space given below.
   ii) Compare Your Answer with those given at the end of the unit.

a) Explain the logic behind flying message application?
   --------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
   ----

b) How will you load the picture at runtime in the image control?
   --------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
   ----

c) Explain various uses of Image control?
   --------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
   ----

d) How can you develop Flying image application so that image is flying across screen?
   --------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
   ----
2.8 Let Us sum up

In this unit you have seen the properties and events related with ScrollBar control, Timer Control and Image Control. You have also seen the practical applications of these controls with the help of programs and codes given in this unit. You have seen the practical applications of Timer control and how its interval property is used in the Timer event to execute code in the Timer event at particular intervals. You have seen how to trap ScrollBar values by reading its value property and using these values in the program. You have also got knowledge of Image control so that you can load picture in the image control at Design time or run time.

2.9 Check your progress: the key
Check Your Progress 1

a) SmallChange is the variation in value you get when clicking on the scrollbar arrows. LargeChange is the variation you get when you click on either side of the ScrollBar indicator.

b) Scroll Event fires when we drag the indicator.
   The Change event fires when you click on the ScrollBar arrows or when you drag the indicator.

c) When a ScrollBar control has focus we can move the indicator using Left, Right, Up, Down, PgUp, PgDown, Home and End Keys.
d) Private Sub VScroll1_Change()
    VScroll2.Value = VScroll1.Value
End Sub

Private Sub VScroll2_Change()
    VScroll1.Value = VScroll2.Value
End Sub

Check Your Progress 2

a) Interval property stands for the number of milliseconds between subsequent pulses on which the Timer event Fires. With the help of Enabled property you can activate or deactivate events.

b) We can trap the pulses send by timer control with the help of Timer event Procedure

c) Take one label control on a form and one timer control. Then write following piece of code for timer event procedure of timer control.

Private sub Timer1_Timer()
    Dim str as String
    Str= Time$
    lblTime.Caption=Str
End sub

Check Your Progress 3

a) The flying message application lets you to fly(move)objects like Label controls ,PictureBox controls, Image controls etc across the form (screen). So to implement this application we have to constantly and continuously move these objects across the screen .For this we have set the Interval property to any appropriate value and then write the code in the Timer event procedure so that each time when the timer event fires ,it constantly moves objects with the help of Left and Top Properties of objects.

b) you can load the image at run time with the help of LoadPicture function Like this:
    Image1.Picture=LoadPicture ('C:\Windows\image1.gif')
c) To load and show bitmaps and Gif and Jpeg images at runtime on a form. They are also used as graphical buttons and toolbars because they support all the usual mouse buttons.

d)

```vbnet
Dim t, l As Integer
Private Sub Form_Load()
    Timer1.Interval = 1
    Form1.Width = 8000
    Form1.Height = 8000
    Image1.Width = 2000
    Image1.Height = 800
    t = 20
    l = 20
End Sub

Private Sub Timer1_Timer()

    If Image1.Top <= 0 Then
        t = 20
    End If

    If Image1.Left <= 0 Then
        l = 20
    End If

    If Image1.Top >= Form1.Height – Image1.Height Then
        t = -20
    End If

    If Image1.Left >= Form1.Width – Image1.Width Then
        l = -20
    End If

    Image1.Top = Image1.Top + t
    Image1.Left = Image1.Left + l

End Sub
```

2.10 The References

6. Steven Holzner, Visual Basic 6 Black Book
7. Mastering VB 6.0 (BPB)
8. Peter Norton, Visual Basic 6
9. Gary Cornell, Visual Basic from ground up(TMH)
10. VB 6 The Complete Reference.
Unit IV
Examples of Enhanced Control

Unit Contains

4.0 Introduction
4.1 Objective
4.2 Example of Rich Text Box
  4.2.1 Previewing a document with copy and searching operation
4.3 Example of Key State Control
  4.3.1 Moving Circle Application
  4.3.2 Keyboard Event Handler Application
4.4 Example of Status bar
  4.4.1 Changing Style of Status bar Application.
  4.4.2 Creating and Using Panels in Status Bar Application
4.5 Example of Slider Control
  4.5.1 Slider for color change Application
4.6 Example of Progress Bar
  4.6.1 Time Display using Progress Bar Application
4.7 Check Your Progress
4.8 Let Us Sum Up
4.9 Check Your Progress: the key
4.10 References

4.0 Introduction
Till Now you have read about many basic controls and enhanced control of visual basic. To run any program in visual basic firstly you have to design it in proper format in design window. You have to align all controls with each other in the form. Then select the control and set its property in the property window. When you have designed all controls on the form, you can start writing code. To write code for any event first select control and then select its event in the code window. When you have written all the required code, you can run the program by pressing the F5 Key.

4.1 Objective
The main aim of this unit is to improve your knowledge about visual basic enhanced control. After going through this unit you will be able to:
  • Design interface and writing code for the program.
Know how to open a file in Rich Text box control, and how to use drive, directory and file list box together.

Learn to handle keystrokes occurs on the form.

Create panels on status bar and changing style of panels.

Use slider control to increase or decrease color on picture box.

Implement the progress bar with timer control to show current time.

4.2 Example of Rich Text Box

Rich Text box is a tool given in visual basic to support files of rtf and txt extension. You can easily open and create files of these formats with this control.

4.2.1 Previewing a document with copy and searching operation

Description: This application help’s you to preview any text or rich text document. You can search any required document by selecting drive in Drive List box and then selecting directory in the directory list box of source section. All documents in selected folder will be displayed in File List box. To see preview of any document first you have to select it from File List Box and then Click Preview Button. To copy file you have to select file from file list box of source section. Type name of new file in textbox of destination section. Destination location can be selected by Drive list box and Directory list box of destination section. To copy file click on Copy button.

To create this application perform following steps

1. Start Microsoft Visual Basic and create a Standard EXE application
2. Save the application in a new folder named **RTF Application**.
3. Save the form as **FrmRTF** and save the project as **RTFApplication**.
4. Add following control to the form

<table>
<thead>
<tr>
<th>Control</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>Name:-FrmRTF</td>
</tr>
<tr>
<td></td>
<td>Border Style: 3 – FixedDialog</td>
</tr>
<tr>
<td></td>
<td>Caption : RTF Application for Preview and Copy Operation.</td>
</tr>
<tr>
<td>Drive List Box</td>
<td>Name : Drivelistbox</td>
</tr>
<tr>
<td>Drive List Box</td>
<td>Name : Drivelistbox2</td>
</tr>
<tr>
<td>Dir List Box</td>
<td>Name : Dirlistbox</td>
</tr>
<tr>
<td>Dir List Box</td>
<td>Name : Dirlistbox2</td>
</tr>
<tr>
<td>File List Box</td>
<td>Name : Filelistbox</td>
</tr>
<tr>
<td></td>
<td>Pattern : &quot;<em>.txt;</em>.rtf&quot;</td>
</tr>
<tr>
<td>Rich Text Box</td>
<td>Name : Richtextbox1</td>
</tr>
</tbody>
</table>
5. Design it as follows:

| Command Button | Name: CmdPreview  
| Caption: &Preview  
| Command Button | Name: Cmdcopy  
| Caption: &Copy  
| Label | Name: LblSource  
| Caption: Source Section  
| Label | Name: LblDest  
| Caption: Destination Section  
| Text Box | Name: TxtDestFName  
| Caption: Type Destination File Name Here

6. Implement the events as follows:

```vba
Private Sub cmdcopy_Click()
    Dim SourceFileName As String
    Dim DestFileName As String
    On Error GoTo err:
    SourceFileName = FileListBox.Path & "\" & FileListBox.FileName
    If TxtDestFName.Text = "" Then
        MsgBox "Type Destination file name", , "Warning"
        Exit Sub
    End If
```

262
DestFileName = DirListBox2.Path & "\" & TxtDestFName.Text
FileCopy SourceFileName, DestFileName
MsgBox "FILE COPY OPERATION COMPLETED", , "Successful"
Exit Sub
err:
MsgBox "Either Source Or Destination is Invalid", , "Warning"
End Sub
Private Sub CmdPreview_Click()
Dim fname As String
On Error GoTo err
fname = FileListBox.Path & "\" & FileListBox.FileName
RichTextBox1.LoadFile fname
Exit Sub
err:
    MsgBox "No File Selected for Preview", , "Warning"
End Sub
Private Sub DirListBox_Change()
FileListBox.Path = DirListBox.Path
If FileListBox.ListCount <> 0 Then
    FileListBox.Selected(0) = True
End If
End Sub
Private Sub DriveListBox_Change()
DirListBox.Path = DriveListBox.Drive
End Sub
Private Sub DriveListBox2_Change()
DirListBox2.Path = DriveListBox2.Drive
End Sub
Private Sub Form_Load()
DirListBox_Change
On_Load_settings
End Sub
Private Sub On_Load_settings()
LblSource.Caption = "Source Section"
LblDest.Caption = "Destination Section"
FileListBox.Pattern = "*.txt;*.rtf"
FrmRTF.Caption = "RTF Application for Preview and Copy Operation"
CmdPreview.Caption = "&Preview"
CmdCopy.Caption = "&Copy"
RichTextBox1.Text = ""
TxtDestFName.Text = ""
End Sub

7. Run the application.

4.3 Example of Key State Control
Whenever you press a key on the keyboard, compute triggers a keyboard event. We can handle these events with visual basic.

4.3.1 Moving Circle Application
Description: This application will move a circle in the form on pressing the UP key, Down Key, Left Key and Right Key. The circle will move in appropriate direction depending on which key is pressed. Only one control the Shape is used in this application. To create this application performs following steps:

1. Start Microsoft Visual Basic and create a Standard EXE application
2. Save the application in a new folder named Moving Circle Application.
3. Save the form as frmMCircle and save the project as MovingCircleApp.
4. Add following control to the form

<table>
<thead>
<tr>
<th>Control</th>
<th>Properties</th>
</tr>
</thead>
</table>
| Form    | Name:FrmMCircle  
|         | Border Style: 3 – FixedDialog  
|         | Caption : Moving Circle Application. |
| Shape1  | Name : Drivelistbox  
|         | Shape : 3 Circle  
|         | BackStyle : 1  
|         | BackColor : &H80000001 |

5. Design it as follows :

6. Implement the events as follows :

   Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
   Select Case KeyCode
   Case vbKeyUp
     Shape1.Top = Shape1.Top - 30
   Case vbKeyDown
Shape1.Top = Shape1.Top + 30
Case vbKeyLeft
Shape1.Left = Shape1.Left - 30
Case vbKeyRight
Shape1.Left = Shape1.Left + 30
End Select
End Sub
Private Sub Form_Load()
Shape1.BackStyle = 1
Shape1.BackColor = &H80000001
End Sub

7. Run the application.

**4.3.2 Keyboard Event Handler Application**

Description: This program gives you idea about handling keyboard. Only a Textbox is used in this application. This program handles all keystrokes performed by user in the Textbox. When a key is pressed it will display a message box which show that what key is pressed and what is the ASCII value of that key. It also handles Shift, Ctrl, Alt key and there combination. To create this application perform following steps:

1. Start Microsoft Visual Basic and create a Standard EXE application
2. Save the application in a new folder named **Keyboard Handler Application**.
3. Save the form as **frmKb** and save the project as **KBHandlerApp**.
4. Add following control to the form

<table>
<thead>
<tr>
<th>Control</th>
<th>Properties</th>
</tr>
</thead>
</table>
| Form        | Name: FrmKb
Border Style: 3 – FixedDialog  
Caption : Keyboard Event Handler Application |
| Text Box    | Name : Text1  
Text : “” |
| Label       | Name : Label1  
Caption : Type In Following Text Box |

5. Design it as follows :
6. Implement the events as follows:

   Private Sub Form_Load()
   Text1.Text = ""
   Label1.Caption = "Type In Following Text Box"
   End Sub

   Private Sub Text1_KeyDown(KeyCode As Integer, Shift As Integer)
       ShiftKey = Shift And 7
       Select Case ShiftKey
           Case 1 ' or vbShiftMask
               MsgBox "You pressed the SHIFT key."
           Case 2 ' or vbCtrlMask
               MsgBox "You pressed the CTRL key."
           Case 4 ' or vbAltMask
               MsgBox "You pressed the ALT key."
           Case 3
               MsgBox "You pressed both SHIFT and CTRL."
           Case 5
               MsgBox "You pressed both SHIFT and ALT."
           Case 6
               MsgBox "You pressed both CTRL and ALT."
           Case 7
               MsgBox "You pressed SHIFT, CTRL, and ALT."
       End Select

       Select Case KeyCode
           Case vbKeyCapital
               MsgBox "Caps Lock key pressed"
           Case vbKeyReturn
               MsgBox "Ener Key is pressed"
           Case vbKeyTab
               MsgBox "Tab key is pressed"
           Case vbKeyF1
               MsgBox "Function key F1 is pressed"
               'All functions keys can be caught in same way
           Case vbKeyHome
               MsgBox "Home key is pressed"
           Case vbKeyEnd
               MsgBox "End key is pressed"
Case vbKeySpace
    MsgBox "Spacebase key is pressed"
Case vbKeyUp
    MsgBox "up key is pressed"
Case vbKeyDown
    MsgBox "Down key is pressed"
Case vbKeyBack
    MsgBox "Backspace key is pressed"
Case vbKeyPageUp
    MsgBox "Page Up key is pressed"
Case vbKeyPageDown
    MsgBox "Page Down key is pressed"
Case vbKeyInsert
    MsgBox "Insert Key is pressed"
Case vbKeyNumlock
    MsgBox "NumLock key is pressed"
Case vbKeyAdd
    MsgBox "Plus Sign (+) key on Number Pad is pressed"
Case vbKeySeparator
    MsgBox "Enter Key on Number pad is pressed"
Case vbKeySubtract
    MsgBox "Minus Sign (-) key on Number Pad is pressed"
Case vbKeyDecimal
    MsgBox "Decimal Point (.) Key on Number Pad is pressed"
Case vbKeyMultiply
    MsgBox "Multiplication Sign (*) key on Number Pad is pressed"
Case vbKeyDivide
    MsgBox "Division Sign (/) key on Number Pad is pressed"
End Select
End Sub
Private Sub Text1_KeyPress(KeyAscii As Integer)
    MsgBox "You have pressed Key: " & Chr(KeyAscii) & " AND ASCII VALUE IS:" & KeyAscii
End Sub

4.4 Example of Status bar
With the StatusBar, you can easily display information about the date, time, and other details about the application and the environment to the user.

4.4.1 Changing Style of Status bar Application.

Description: This application show how we can change style of a status bar. Six command buttons are in this application and on clicking on that, style of status bar will changes, for example when you click on Caps Lock Key button then status bar will show you current status of caps key. To create this application perform following steps:

1. Start Microsoft Visual Basic and create a Standard EXE application
2. Save the application in a new folder named StatusBarStyle Application.
3. Save the form as **FrmStatubar** and save the project as **StatusbarStyleApp**.

4. To use the needed Status bars, on the main menu, click Project -> Components...

5. In the Components dialog box, click the Microsoft Windows Common Controls 6.0 (SP4) check box

6. Add following control to the form

<table>
<thead>
<tr>
<th>Control</th>
<th>Properties</th>
</tr>
</thead>
</table>
| Form             | Name: FrmStatubar  
|                  | Border Style: 3 – FixedDialog  
|                  | Caption: Changing Style of Status bar Application |
| Command Button   | Name: CmdDate  
|                  | Caption: Current Date |
| Command Button   | Name: CmdTime  
|                  | Caption: Type In Following Text Box |
| Command Button   | Name: CmdCaps  
|                  | Caption: Caps Lock Key |
| Command Button   | Name: CmdNum  
|                  | Caption: Num Lock Key |
| Command Button   | Name: CmdIns  
|                  | Caption: Insert Key |
| Command Button   | Name: CmdScrl  

7. Design it as follows

<table>
<thead>
<tr>
<th>Status Bar</th>
<th>Caption : Scroll Lock Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name : StatusBar1</td>
<td></td>
</tr>
<tr>
<td>Align : 2- vbAlignBottom</td>
<td></td>
</tr>
</tbody>
</table>

8. Implement the events as follows

```vbnet
Private Sub On_load_setting()
    CmdDate.Caption = "Current Date"
    CmdTime.Caption = "Current Time"
    CmdCaps.Caption = "Caps Lock Key"
    CmdNum.Caption = "Num Lock Key"
    CmdIns.Caption = "Insert Key"
    CmdScrl.Caption = "Scroll Lock Key"
End Sub
Private Sub CmdCaps_Click()
    StatusBar1.Panels(1).Style = sbrCaps
End Sub
Private Sub CmdDate_Click()
    StatusBar1.Panels(1).Style = sbrDate
End Sub
Private Sub CmdIns_Click()
    StatusBar1.Panels(1).Style = sbrIns
End Sub
Private Sub CmdNum_Click()
    StatusBar1.Panels(1).Style = sbrNum
End Sub
Private Sub CmdScrl_Click()
    StatusBar1.Panels(1).Style = sbrScrl
End Sub
Private Sub CmdTime_Click()
    StatusBar1.Panels(1).Style = sbrTime
End Sub
```
Private Sub Form_Load()
FrmStatubar.Caption = "Changing Style of Status Bar Application"
LblStatusbar.Caption = "Click on Command Button To Change Style
of Status Bar"
Call On_load_setting
End Sub

9. Run the application.

4.4.2 Creating and Using Panels in Status Bar Application
Description: This program will show you how you can create different panels
in the status bar. In this application Panels are created. Four Textbox's and
four command buttons are used for displaying text in different panels. You
have to type text in text box and then click on its related command button, the
typed text will be displayed in corresponding panel of status bar. To create this
application perform following steps

1. Start Microsoft Visual Basic and create a Standard EXE application
2. Save the application in a new folder named StatusBarPanel
   Application.
3. Save the form as FrmStatubarP and save the project as
   StatusBarPanelApp.
4. To use the needed status bars, on the main menu, click Project ->
   Components...
5. In the Components dialog box, click the Microsoft Windows Common
   Controls 6.0 (SP4) check box
6. Add following control to the form

<table>
<thead>
<tr>
<th>Control</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>Name:- FrmStatubarP&lt;br&gt;Border Style: 3 – FixedDialog&lt;br&gt;Caption : Creating and Using Panels Applications</td>
</tr>
<tr>
<td>Command Button</td>
<td>Name : CmdPanel1&lt;br&gt;Caption : Click To Display on Panel 1</td>
</tr>
<tr>
<td>Command Button</td>
<td>Name : CmdPanel2&lt;br&gt;Caption : Click To Display on Panel 2</td>
</tr>
<tr>
<td>Command Button</td>
<td>Name : CmdPanel3&lt;br&gt;Caption : Click To Display on Panel 3</td>
</tr>
<tr>
<td>Command Button</td>
<td>Name : CmdPanel4</td>
</tr>
</tbody>
</table>
7. Design it as follows

<table>
<thead>
<tr>
<th>Text Box</th>
<th>Name</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TxtPanel1</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td></td>
<td>TxtPanel2</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td></td>
<td>TxtPanel3</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td></td>
<td>TxtPanel4</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>

8. Implement the events as follows

```vba
Private Sub CmdPanel1_Click()
    StatusBar1.Panels(1).Text = TxtPanel1.Text
End Sub

Private Sub CmdPanel2_Click()
    StatusBar1.Panels(2).Text = TxtPanel2.Text
End Sub

Private Sub CmdPanel3_Click()
    StatusBar1.Panels(3).Text = TxtPanel3.Text
End Sub

Private Sub CmdPanel4_Click()
End Sub
```

Private Sub Form_Load()
on_load_setting
Dim panel2 As Panel
Dim panel3 As Panel
Dim panel4 As Panel
Set panel2 = StatusBar1.Panels.Add()
Set panel3 = StatusBar1.Panels.Add()
Set panel4 = StatusBar1.Panels.Add()
End Sub
Private Sub on_load_setting()
    LblPanel1.Caption = "Enter Text For Panel 1 Here"
    LblPanel2.Caption = "Enter Text For Panel 2 Here"
    LblPanel3.Caption = "Enter Text For Panel 3 Here"
    LblPanel4.Caption = "Enter Text For Panel 4 Here"
    TxtPanel1.Text = ""
    TxtPanel2.Text = ""
    TxtPanel3.Text = ""
    TxtPanel4.Text = ""
    CmdPanel1.Caption = "Click To Display on Panel 1"
    CmdPanel2.Caption = "Click To Display on Panel 2"
    CmdPanel3.Caption = "Click To Display on Panel 3"
    CmdPanel4.Caption = "Click To Display on Panel 4"
End Sub

9. Run the application.

4.5 Example of Slider Control
The Slider control provides a means for users to enter numeric data into a program.

4.5.1 Slider for color change Application
Description: This application will give you idea of using Slider control on your form. It uses three slider control and one picture box. First slider for red color, second slider for green color and third slider to blue color. Picture box is used to show you color changes made by combination of these there slider’s. To create this application perform following steps

1. Start Microsoft Visual Basic and create a Standard EXE application
2. Save the application in a new folder named SliderForColor Application.
3. Save the form as FrmSliderFcolor and save the project as SliderForColorApp.
4. To use the needed Slider control, on the main menu, click Project -> Components...
5. In the Components dialog box, click the Microsoft Windows Common Controls 6.0 (SP4) check box
6. Add following control to the form

<table>
<thead>
<tr>
<th>Control</th>
<th>Properties</th>
</tr>
</thead>
</table>

272
7. Design it as follows

<table>
<thead>
<tr>
<th>Form</th>
<th>Name: FrmSliderFcolor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Border Style: 3 – FixedDialog</td>
</tr>
<tr>
<td></td>
<td>Caption: Slider For Color Change</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Picture Box</th>
<th>Name: pctpreview</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Slider</th>
<th>Name: SldRed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min: 0</td>
<td>Max: 255</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Slider</th>
<th>Name: SldGreen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min: 0</td>
<td>Max: 255</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Slider</th>
<th>Name: SldBlue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min: 0</td>
<td>Max: 255</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Label</th>
<th>Name: lblred</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caption: Red</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Label</th>
<th>Name: lblgreen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caption: Green</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Label</th>
<th>Name: lblblue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caption: Blue</td>
<td></td>
</tr>
</tbody>
</table>

8. Implement the events as follows

```vbnet
Private Sub Form_Load()
    pctpreview.BackColor = RGB(128, 128, 128)
    SldRed.Value = 127
    SldGreen.Value = 127
    SldBlue.Value = 127
    on_load_setting
End Sub
Private Sub SldBlue_Change()
    Dim ValueRed As Integer
    Dim ValueGreen As Integer
    Dim ValueBlue As Integer
```
ValueRed = SldRed.Value
ValueGreen = SldGreen.Value
ValueBlue = SldBlue.Value
pctpreview.BackColor = RGB(ValueRed, ValueGreen, ValueBlue)
End Sub
Private Sub SldBlue_Scroll()
SldBlue_Change
End Sub
Private Sub SldGreen_Change()
    Dim ValueRed As Integer
    Dim ValueGreen As Integer
    Dim ValueBlue As Integer
    ValueRed = SldRed.Value
    ValueGreen = SldGreen.Value
    ValueBlue = SldBlue.Value
    pctpreview.BackColor = RGB(ValueRed, ValueGreen, ValueBlue)
End Sub
Private Sub SldGreen_Scroll()
SldGreen_Change
End Sub
Private Sub sldred_Change()
    Dim ValueRed As Integer
    Dim ValueGreen As Integer
    Dim ValueBlue As Integer
    ValueRed = SldRed.Value
    ValueGreen = SldGreen.Value
    ValueBlue = SldBlue.Value
    pctpreview.BackColor = RGB(ValueRed, ValueGreen, ValueBlue)
End Sub
Private Sub sldred_Scroll()
    sldred_Change
End Sub
Private Sub on_load_setting()
    lblred.Caption = "Red"
    lblgreen.Caption = "Green"
    lblblue.Caption = "Blue"
End Sub

9. Run the application.

4.6 Example of Progress Bar

Progress bar shows that your application is busy with some work. With this bar user can get the idea about that how much more time will be needed to complete the work.

4.6.1 Time Display using Progress Bar Application
Description: This application will show System Time in the form of Progress bar. Three Progress bars are used for this purpose. First progress bar is used for hours, second progress bar is used for minutes and third progress bar is used for seconds. One timer control is attached with progress bar. To create this application perform following steps:

1. Start Microsoft Visual Basic and create a Standard EXE application
2. Save the application in a new folder named **TimeDisplayPrgBar Application**.
3. Save the form as **FrmPrgTime** and save the project as **TimeDisplayPrgBarApp**.
4. To use the needed progress bars, on the main menu, click Project -> Components...
5. In the Components dialog box, click the Microsoft Windows Common Controls 6.0 (SP4) check box
6. Add following control to the form

<table>
<thead>
<tr>
<th>Control</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>Name:-FrmPrgTime &lt;br&gt;Border Style: 3 – FixedDialog &lt;br&gt;Caption : Time Display Using Progress Bar</td>
</tr>
<tr>
<td>Progress Bar</td>
<td>Name : pgrhours &lt;br&gt;Min : 0 &lt;br&gt;Max : 23</td>
</tr>
<tr>
<td>Progress Bar</td>
<td>Name : pgrminutes &lt;br&gt;Min : 0 &lt;br&gt;Max : 59</td>
</tr>
<tr>
<td>Progress Bar</td>
<td>Name : pgrseconds &lt;br&gt;Min : 0 &lt;br&gt;Max : 59</td>
</tr>
<tr>
<td>Label</td>
<td>Name :  lblhours</td>
</tr>
<tr>
<td>Label</td>
<td>Name :  lblminutes</td>
</tr>
<tr>
<td>Label</td>
<td>Name :  lblseconds</td>
</tr>
<tr>
<td>Label</td>
<td>Name :  LblH &lt;br&gt;Caption : Hours</td>
</tr>
<tr>
<td>Label</td>
<td>Name :  LblM</td>
</tr>
</tbody>
</table>
7. Design it as follows

<table>
<thead>
<tr>
<th>Decription</th>
<th>Caption: Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>Name: LblS</td>
</tr>
<tr>
<td></td>
<td>Caption: Seconds</td>
</tr>
<tr>
<td>Time Control</td>
<td>Name: Timer1</td>
</tr>
<tr>
<td></td>
<td>Interval: 20</td>
</tr>
</tbody>
</table>

8. Implement the events as follows

```vba
Private Sub Form_Load()
on_load_setting
End Sub
Private Sub Timer1_Timer()
    Dim CurTime As Date
    Dim ValHours As Integer
    Dim ValMinutes As Integer
    Dim ValSeconds As Integer
    CurTime = Time
    ValHours = Hour(CurTime)
    ValMinutes = Minute(CurTime)
    ValSeconds = Second(CurTime)
    pgrhours.Value = ValHours
    pgrminutes.Value = ValMinutes
    pgrseconds.Value = ValSeconds
    lblhours.Caption = CStr(ValHours)
    lblminutes.Caption = CStr(ValMinutes)
    lblseconds.Caption = CStr(ValSeconds)
End Sub
Private Sub on_load_setting()
    FrmPrgTime.Caption = "Time Display Using Progress Bar"
    LblH.Caption = "Hours"
    LblM.Caption = "Minutes"
    LblS.Caption = "Seconds"
```
Timer1.Interval = 20
End Sub

9. Run the application.

4.7 Check Your Progress

Notes: i) Write your answers in the space given below.
   ii) Compare Your Answer with those given at the end of the unit.

a) Write code to restrict input in textbox to numeric entry only.
   
   b) Write steps to insert status bar on the form?

   c) Write code to convert capital Letter to small letters.

   d) Write visual basic constants for Caps lock, Num lock, F1 function key, Home key and page up key.

4.8 Let Us Sum Up

After going through this unit, you would have achieved the objectives stated earlier in the Unit. Let us recall what we have discussed so far.

To create a visual basic program you have to design its interface in design window. First of all you have to select the required controls from toolbox and drag them at required positions on the form. Then set properties of controls with property window. To write code for any event first select control and then select its event in the code window. After writing all needed code, press F5 key to run your application.

Visual basic has two types of controls: basic controls and advanced controls. Basic controls are directly available on toolbox. To select advance control first
click on Project Menu, then select components, it will show Component dialog box. Then select required control from the list.
You have created following applications:
- An application in which you learned how drive list box, directory list box and file list box can work together with Rich Text Box.
- Applications which show you different keyboard events and how to handle these events.
- Applications which show you how to change style of a status bar and how to create different panel on status bar.
- Application to use three slider controls for controlling color in a picture box.
- Application which shows your computer current time in the form of three progress bar, one for Hour, one for minute and one for second.

4.9 Check Your Progress: The Key

(a) Private sub Text1_KeyPress (KeyAscii as Integer)
If (keyAscii >= 48 And keyAscii <= 57) Then
KeyAscii = 0
End if
End Sub

(b) Click on the Project menu, and then click Components... option. It will display Components dialog box. In this dialog box select Microsoft Windows Common Controls 6.0 (SP4) check box.

(c) Private sub Text1_KeyPress (KeyAscii as Integer)
If (keyAscii >= 65 And keyAscii < 97) Then
KeyAscii = KeyAscii + 32
End if
End Sub

(d) vbKeyCapital
vbKeyNumlock
vbKeyF1
vbKeyHome
vbKeyPageUp

4.10 References

- Steven Holzner, “Visual Basic 6 Black Book”, The Coriolis Group
- Peter Wright, “Beginning Visual Basic 6”, Springer-Verlag New York Inc
- http://www.vb6.us/
- http://www.vbtutor.net/
- http://www.visualbasic6class.com/