VISUAL BASIC 6.0 OVERVIEW

GENERAL CONCEPTS

Visual Basic is a **visual** programming language. You create forms and controls by drawing on the screen rather than by coding as in traditional languages. *Visual Basic* is **object-oriented**. You create objects (forms and controls) on the screen and then control them by using properties, procedures and methods. *Visual Basic* is a programming environment which allows you to write **event-driven** applications. This is different from the traditional programming environment. When you program an event-driven application, your program responds to particular events such as a key press on the keyboard or a click of a mouse button. Such applications are usually simple to use.

Recall that a *Visual Basic* application's interface is made up of objects (forms and controls). Each object recognizes a user's actions such as pressing keys on the keyboard or clicking on a mouse button. These actions are referred to as **events**. Each form and control in *Visual Basic* responds to a pre-defined set of events such as a key press or a mouse click.

When an event occurs in your application, *Visual Basic* automatically recognizes the event and runs the code that you have written for it. This code is called an **event procedure**. An event procedure is made up of an object name and an event name (note that an object can handle multiple events). The object name for all forms is **Form**. The object name for other controls is the control's **Name** property. Remember that an event procedure can have one or more statements. Event procedure names consist of the control name, an underscore and the event name. For example, the event procedure name for the **Click** event of the **cmdOK** command button would be **cmdOK_Click**.

```
Sub cmdOK_Click ()
    lblWelcome.Caption = "Welcome to Visual Basic Programming"
End Sub
```

To access a control and its properties in code, you use the *object.property* notation. For example, to change the *Caption* property of the *lblWelcome* label, you would use the following statement:

```
lblWelcome.Caption = "Welcome to Visual Basic Programming"
```

This statement will display the message *Welcome to Visual Basic Programming* in the *lblWelcome* label.

An event procedure runs only when the event occurs, and your application remains idle until that happens. When you click on an *OK* button for example, the code in the *cmdOK_Click* procedure will be executed; this code will be executed every time the *OK* button is clicked. Therefore, you need to write the code only for the events that you want your application to respond to. In order to determine what events you need to write code for, you have to think about what the user may do and how you want your program to respond.

Event procedures can also
- trigger other event procedures;
- change an object's properties; or
- call other general procedures that are not tied to any event.

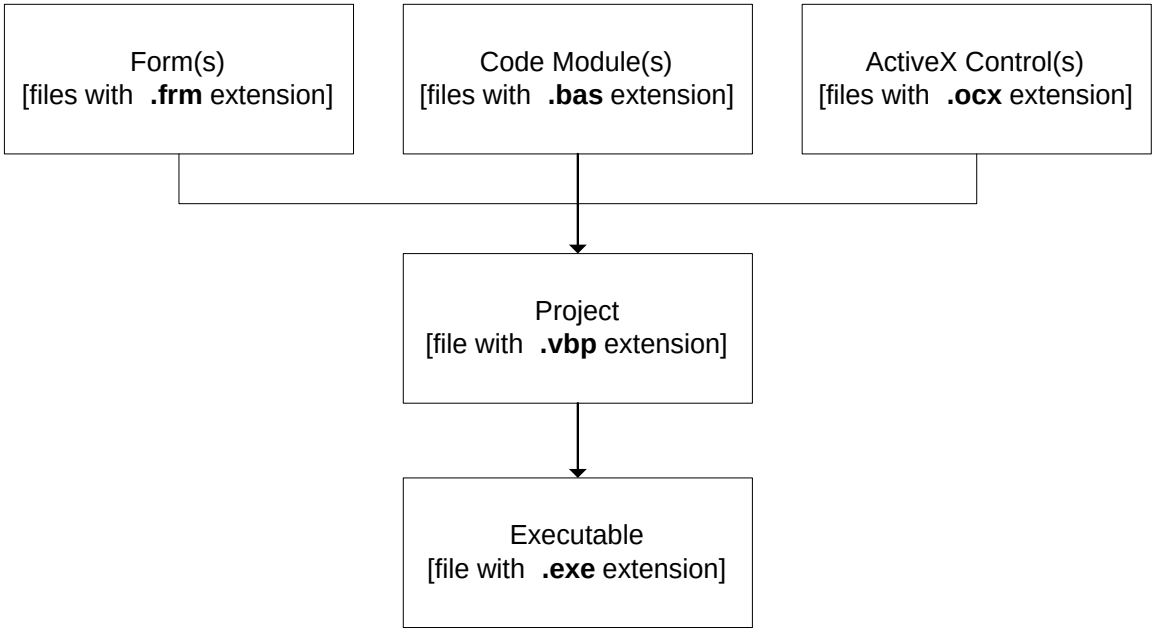
STRUCTURE OF A VISUAL BASIC PROJECT

An application in *Visual Basic* is more commonly known as a **project**. A project includes:

- Forms: Windows, backgrounds and dialog boxes;
- Controls: Any graphical objects placed on the forms;
- Code: The source code (declarations and procedures).

There are three steps involved when creating an application:
1. Creating the interface
2. Setting the properties
3. Writing the source code

Every project contains at least one form. You can add as many additional forms as your application requires. A *Visual Basic* project has the following structure:



The first form of the application that is created is called the **startup form**. When an application is executed (started), the form is loaded first. Therefore, the code that is included (if any), in the **Form_Load** procedure is executed first. Typically, the load event procedure is used to include initialization code for a form; for example, specifying default settings for controls, initializing form-level variables and placing contents into list boxes or combo boxes.

An example might look similar to:

```
Sub Form_Load ()
    Students = 0
    chkSchool.Value = 1
    optDayTime.Value = True
End Sub
```

DIFFERENCES FROM VISUAL BASIC 3.0

While *Visual Basic 6* is very similar to version 3, there are some differences that are worth mentioning.

- One of the first differences that you will encounter is the extension of some of the project files. As you see in the diagram shown above, when you save a project, it will now have an extension of **VBP** (as opposed to MAK in version 3). Also, additional controls have an OCX extension (similar to the VBX files in version 3). In addition, when you run a project to test it, a workspace file (with an extension of VBW) will be created in the directory where the program resides. This file can be deleted but will be recreated anytime you run the program.
- Another main difference that you should be aware of is that you can now use long filenames for your project files (recall that in version 3, you were limited to filenames that could not be longer than 8 characters).
- Version 6 has many more additional controls and powerful features than version 3 that we will learn about as we encounter them.
- Although any code that works in version 3 will work in version 6, this later version cannot open projects saved in version 3. Please ensure that you use version 6 of *Visual Basic* from this point on to avoid complications.

VARIABLES
As you recall, a variable is a memory location capable of containing a value that can be modified during the execution of a program. Each variable has a unique name that identifies it. In order for a variable name to be valid, certain rules must be followed:

- variable names must begin with a letter;
- variable names must only contain letters, numerical digits, and the underscore character (_);
- variable names cannot be reserved words (words that have a special meaning to VB).

Note that variable names are case insensitive (for example, *homeroom* and *HOMEROOM* are equivalent; however, *home_room* is different).

DATA TYPES
Computer languages use many types of data, called **data types**. Some common data types are:

Integer The set of integer numbers (no decimals; e.g. 314, -57, 0, 89). Takes up 2 bytes of storage space and its range is -32,768 to 32,767.

Long The set of long integers (integers that are outside the range given above). Takes up 4 bytes of storage space and its range is -2,147,483,648 to 2,147,483,647.

Single The set of real numbers (single-precision floating point; with a decimal point). Takes up 4 bytes of storage space and its range is -3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values.

Double The set of (double-precision) real numbers. Takes up 8 bytes of storage space and its range is -1.79769313486232E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values.

String Text. When assigning a value to this type, be sure to include the double quotes. For example, "Hello" or "12A". There are two types of string declarations. For a fixed-length string, it takes up 1 byte of storage space per character and its maximum length is approximately 65,500 characters. For a variable-length string, it takes up the length of the string plus 10 bytes of storage space. Its maximum length is approximately 2 billion.

Boolean Takes up 2 bytes of storage space and has the value *True* or *False*.

Declaring a variable means stating its data type and reserving space in memory for the storage of a value using the keywords **Dim** and **As**. For example,

```
Dim Age As Integer
Dim Price As Single
Dim Student_Name As String       ' Variable length string
Dim Home_Form As String * 3      ' Fixed length string - Learn this year
```

Assigning a variable means giving a value to the variable. A variable **must** be assigned before its contents can be used. For example,

```
Age = 16
Price = 7.99
Student_Name = "Bugs Bunny"
Home_Form = "12A"
```

Constants are values that do not change (e.g. PI = 3.1415 or PST = 0.08) throughout the execution of a program. Declaring a constant means that a certain name stands for a constant and has a certain value (the keyword *Const* is used). For example,

```
Const PI = 3.1415
Const PST = 0.08
```

Constants are used because:

- they make the code more readable; and,
- they make programs easier to modify or revise.

Soft coding is using constant names in the code rather than actual values (as opposed to **hard coding**).

SCOPE OF DECLARATIONS

Type	*Scope*	*Keyword(s)*	*Location*
Local	within one procedure	Dim, Const	inside the procedure
Form	within one form	Dim, Const	form's general declarations section
Module	within one code module	Dim, Const	code module's general declarations section
Global	whole application	Global, Global Const	code module's general declarations section

TYPES OF PROCEDURES

Event procedures are invoked by a user action or system event (e.g. **cmdCalculate_Click ()** is invoked by a mouse click on the *cmdCalculate* button).
General procedures are invoked by being called through code (e.g. **Pause X** might be a procedure which pauses the execution of the program for X seconds). We will learn how to write and use general procedures this year.
Methods are built-in procedures that act on objects (e.g. picMain.**Print** can be used to print onto a picture box). We will also learn how to use additional methods this year.

ORDER OF OPERATIONS

Expressions are evaluated from left to right using the following order:

( ) **Brackets** are evaluated first; then,
^ **Exponents**; then,
/ * **Division** and **Multiplication** (in the order that they appear); then,
+ - **Addition** and **Subtraction** (in the order that they appear).

CONDITIONAL STRUCTURES

Boolean expressions have only one value (*True* or *False*). They are used in decision making. Multiple conditions can be combined by using the **And** and **Or** Boolean operators. For example,

$condition_1$ **And** $condition_2$	is True if both conditions are True
$condition_1$ **Or** $condition_2$	is True if at least one of the conditions is True

An **If ... Then ... Else** structure allows conditional execution of code based on the evaluation of an expression. The syntax for the use of this structure is:

```
If condition1 Then
   {instruction(s) to be executed if condition1 is true}
ElseIf condition2 Then
   {instruction(s) to be executed if condition2 is true}
ElseIf condition3 Then
   {instruction(s) to be executed if condition3 is true}
Else
   {instruction(s) to be executed if none of the conditions are true}
End If
```

When evaluating a single expression that has several possible actions, a **Select Case** structure may be more useful. We will discuss this structure in more detail later in this course. Note that when a condition has been satisfied (found to be true), the rest of the conditions are bypassed.

PRE-DEFINED FUNCTIONS

		Example:
Abs(x**)**	calculates and returns the absolute value of *x*	`Abs(-5) = 5`
Int(x**)**	calculates and returns the largest integer that is smaller than *x*	`Int(8.9) = 8`
x **Mod** y	calculates and returns the modulus or remainder of *x / y*	`9 mod 4 = 1`
x **** y	calculates and returns the integer result of *x / y*	`9 \ 4 = 2`
Asc(s**)**	returns the ANSI[1] numeric value of the first character in *s*	`Asc("A") = 65`
Chr$(x**)**	returns the character of *x* (as defined in the ANSI table)	`Chr$(65) = "A"`
Str$(x**)**	converts *x* to a string and returns it	`Str$(123) = "123"`
Val(s**)**	converts *s* to a number (if possible) and returns it	`Val("123") = 123`
LCase$(s**)**	returns the lower case equivalent of string *s*	`LCase$("HELLO") = "hello"`
UCase$(s**)**	returns the upper case equivalent of string *s*	`UCase$("Hello") = "HELLO"`
Len(s**)**	returns the length of the string *s*	`Len("Hello") = 5`
Left$(s, l**)**	returns the *l* leftmost characters of string *s*	`Left$("Hello",2) = "He"`
Mid$(s, b, l**)**	returns a substring of string *s*, starting at column *b* and having length *l*	`Mid$("HELLO",2,3)="ELL"`
Right$(s, l**)**	returns the *l* rightmost characters of string *s*	`Right$("HELLO",2) = "LO"`

There is no pre-defined function to **round off** a number. However, you can use the following general formula to round off a number *N* to *d* decimal places:

```
N = Int (N * 10 ^ d + 0.5) / (10 ^ d)
```

Concatenation is the process of combining strings together. The concatenation character is the ampersand (&). For example,

if `A = "Riverdale"` and `B = " CI"`, then the result of `A & B` is `"Riverdale CI"`

DISPLAYING MORE THAN ONE FORM

Some applications require only one form. Very often however, an application needs to display more than one form. The **Show** method is used to display an additional form. To display a form, you use the syntax:

form_name`.`**Show** *Style*

where, *Style* determines whether the form is **modal** or **modeless**. If *Style* is 0, the form is modeless; if *Style* is 1, the form is modal. A modal form is a window which requires the user to take some action before the focus can switch to another form within the same application. A modeless form does not require the user to take any action before the focus can switch to another form within the same application. When a modal form is displayed, no user input can occur in any other form until the modal form is hidden or unloaded; also, any *Visual Basic* code after the Show method will not be executed until the form is hidden or unloaded (the **MsgBox** function is an example of a modal form). Although other forms in your application are disabled when a modal form is displayed, other applications are not.

For example, the code to display a form named *frmCopy* in a modal style, would be:

```
frmCopy.Show 1
```

Visual Basic 6 also has the form constants *vbModal* and *vbModeless* that can and should be used instead of the values 1 and 0 respectively. For example, the last code line shown above is equivalent to:

```
frmCopy.Show vbModal
```

[1] ANSI (American National Standard Institute) is the character set which is used in Windows. Characters 1 through 127 are the same as in the ASCII table; however, characters 128 through 255 are different.

UNLOADING A FORM

Unloading a form or control may be necessary or expedient is some cases where the memory used is needed for something else, or when you need to reset properties to their original values. To unload a form, the syntax is:

Unload *form_name*

For example, to unload a form named *frmCopy*, the code would be:

```
Unload frmCopy
```

HIDING A FORM

Hiding a form or control hides the form (removes it from the screen) but does not unload it (it remains in memory). A hidden form's controls are not accessible by the user but they are available to the running application. To hide a form, the syntax is:

form_name.**Hide**

For example, to hide a form named *frmCopy*, the code would be:

```
frmCopy.Hide
```

LOOPING STRUCTURES

A **For ... Next** structure executes a group of instructions a specific number of times. The syntax for the use of this structure is:

```
For counter = starting_value To ending_value [Step increment]
   {instructions to be executed}
Next counter
```

The *counter* is a numeric variable that is used as a loop counter. The *starting_value* is the initial value that is assigned to the *counter*; the *ending_value* is the final value of the *counter*. The **Step** has to be present if the *increment* value is not 1; if the **Step** option is not present, the *increment* value is 1. The *increment* is added to the *counter* each time the loop is executed. The **Next** statement indicates the end of this type of a looping structure. It adds the *increment* to the *counter* and checks to see if the statements within the loop should be executed again (done as long as the value of *counter* is inclusively between the *starting_value* and *ending_value*).

A **Do ... Loop** structure is another type of loop that we will learn this year. It executes a group of instructions while a certain condition is true. The syntax for the use of this structure may vary depending on the situation:

Syntax 1:

```
Do [{While | Until} condition]
   {instructions to be executed}
Loop
```

Syntax 2:

```
Do
   {instructions to be executed}
Loop [{While | Until} condition]
```

The **Do** must be the first statement in the structure. The **While** indicates that the statements inside the loop should be executed while the *condition* is true (the loop will be executed while the *condition* is true). The **Until** indicates that the statements inside the loop should be executed until the *condition* becomes true (the loop will be executed while the *condition* is false).