LECTURE NOTES

PROGRAMMING WITH VISUAL BASIC 6.0

CHAPTER 6

“USING LOOPS AND TIMERS”

PREPARED BY

Mazhar javed

CHAPTER 6: USING LOOPS AND TIMERS

This chapter describes how to write repeating statements (loops) in program code and how to use the **Timer** control to create clocks and other time-related utilities. In this chapter, you will learn how to:

- Write the **For…Next Loop** statement to run a block of code a set number of times.
- Write the **Do… Loop** statement to run a block of code until a specific condition is met.
- Use the **Timer** control to create a digital clock and other special effects.

6.1: Writing FOR….NEXT Loops

With a **For...Next** loop, you can execute a specific group of program statements in an event procedure a specific number of times. This can be useful when you want to perform several related calculations, work with elements on the screen, or process several pieces of user input.

6.1.1:Anatomy of a FOR...NEXT Loop

A **For...Next** loop is really just a shorthand way of writing out a long list of program statements. Since each group of statements in the list would do essentially the same thing, Visual Basic lets you define a group of statements and request that it be executed as many times as you want.

For...Next Loop Syntax

The syntax for a **For...Next** loop looks like this:

```
For variable = start To end
    statements to be repeated
Next variable
```

In this syntax statement, **For**, **To**, and **Next** are required keywords, and the equal sign (=) is a required operator. First, you replace *variable* with the name of a numeric variable that keeps track of the current loop count. Next, you replace *start* and *end* with numeric values that represent the starting and stopping points for the loop. The line or lines between the **For** and **Next** statements are the instructions that repeat each time the loop executes.

Making a Beeper

For example, the following **For...Next** loop sounds four beeps in rapid succession from the computer's speaker:

```
For i = 1 To 4
    Beep
Next i
```

This loop is the functional equivalent of writing the **Beep** statement four times in a procedure. To the Visual Basic compiler, the loop looks like this:

```
Beep
Beep
Beep
Beep
```

The loop uses the variable *i*. By convention, *i* stands for the first integer counter in a **For...Next** loop. Each time Visual Basic executes the loop, the counter variable increases by one. (The first time through the loop, the variable contains a value of 1, the value of *start*. The last time through, the variable value is 4, the value of *end*.) As you'll see in the following examples, you can use this counter variable in your loops to great advantage.

6.1.2:Using the Counter Variable in a Loop

A counter variable is just like any other variable. You can assign counter variables to properties, use them in calculations, or display them in a program. One of the handiest techniques for displaying a counter variable is to use the **Print** method. This method displays output on a form or prints output with an attached printer. The **Print** method has the following syntax:

```
Print expression
```

where *expression* is a variable, property, text value, or numeric value in the procedure.

Print Method Counter Variable

For example, you could use the **Print** method in a **For…Next** loop to display output on a form:

```
For i = 1 To 10
    Print "Line"; i
Next i
```

The **Print** method displays the word *Line*, followed by the loop counter, 10 times on the form. The **Print** method uses these symbols to separate elements in an expression list:

Symbol	Behavior
comma (**,**)	Places the elements one tab field apart.
semicolon (**;**)	Places elements side by side. (Visual Basic displays the counter variable next to the string with no additional spaces in between.)

If you were to run this program, however, you would probably notice that there *is* a space between “Line” and the counter variable. That's because when the **Print** method prints numeric values, Visual Basic reserves a space for a minus sign, even if the minus sign isn’t needed.

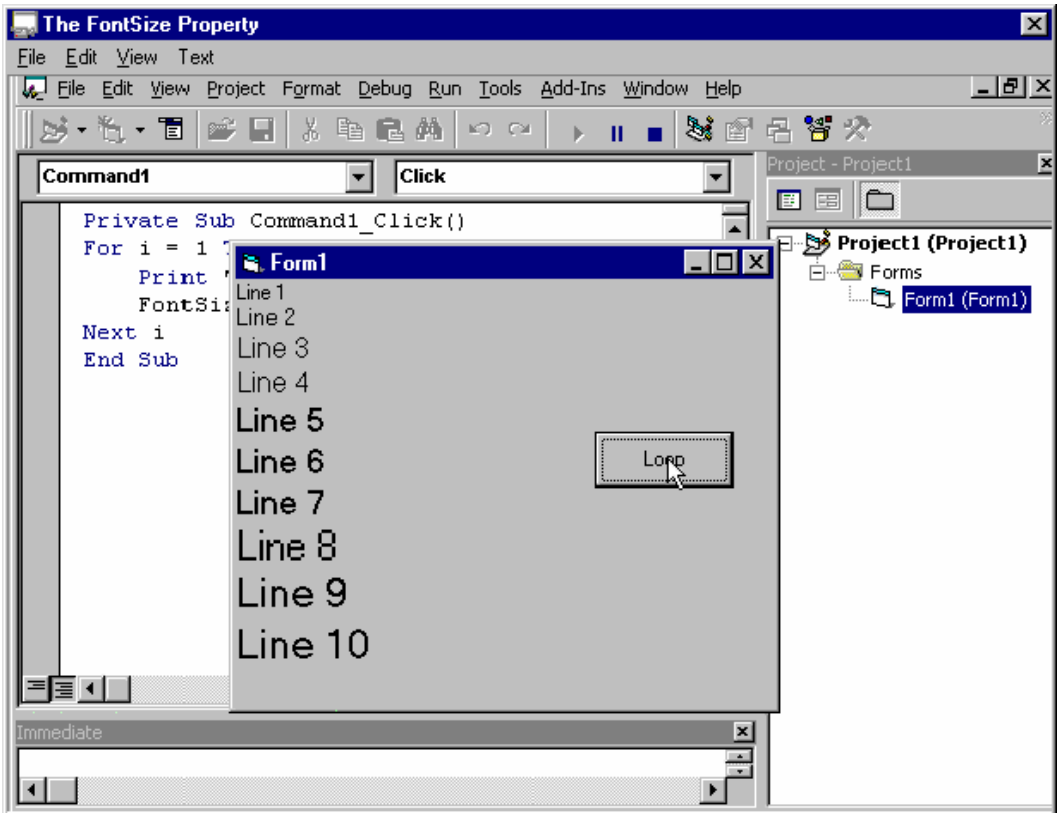
Note You can use any combination of semicolons and commas to separate expression list elements.

6.1.3:Changing Properties with a Loop

In Visual Basic, you can use loops to change properties and update key variables that:

- Open bitmaps with the **Picture** property.
- Keep a running total with a variable.
- Change the text size on your form by updating the form's **FontSize** property.

The example below shows how **a For…Next loop** can change the text size on a form by changing the form's **FontSize** property. (The **FontSize** property adjusts the point size of the text on a form. You can use it as an alternative to changing the point size with the Font property.) In the figure given below, you'll see the program run through the loop to create type that seems to grow as the program runs.



In this example program the only command button on the form is programmed as follows:

```
Private Sub Command1_Click

For I=i To 10
      Print "Line"; i
      FontSize=10 + i
Next I

End Sub
```

When the program runs and whenever the command button `Command1` is clicked, the code written in event procedure writes "Line" statement with a counter variable on the form at a special order. Please see the code and observe the function of For Loop.

6.1.4:Complex FOR...NEXT Loops

Using counter variables in a **For...Next** loop can be a powerful tool in your programs. With a little imagination, you can use counter variables to create several useful sequences of numbers in your loops.

Creating Loops with a Custom Counter

You can create a loop with a counter pattern other than 1, 2, 3, 4, and so on. First, you specify a different start value in the loop. Then, you use the **Step** keyword to increment the counter at different intervals. For example, this loop controls the print sequence of numbers on a form:

```
For i = 5 To 25 Step 5
    Print i
Next i
```

The program prints this sequence of numbers:

```
5
10
15
20
25
```

Specifying Decimal Values

You can also specify decimal values in a loop. For example, here's another **For...Next** loop that controls the print sequence of numbers on a form:

```
For i = 1 To 2.5 Step 0.5
    Print i
Next i
```

The program prints this sequence of numbers:

```
1
1.5
2
2.5
```

Nested For...Next Loops

You can also place one **For...Next** loop inside another to create really interesting effects. If you try this, be sure you use a different counter variable for each loop, such as **i** for the first, and **j** for the second.

6.1.5:Using the EXIT FOR Statement

Finally, you may wish to know how to leave a loop early if the need arises. In Visual Basic, you can use an **Exit For** statement to exit a **For...Next** loop before the loop has finished executing. With this capability, you can respond to specific events that occur before the loop runs the pres et number of times.

For example, in this **For...Next** loop, the loop prompts the user for 10 names and prints them on the form, unless the user enters the word *Done*. (If the user does enter *Done*, the program jumps to the first statement that follows the **Next** statement):

```
For i = 1 To 10
    InpName = InputBox("Type a name or Done to quit.")
    If InpName = "Done" Then Exit For
    Print InpName
Next i
```

As this example shows, you can use If statements with **Exit For** statements. You'll find this combination useful for handling special cases that come up in a loop, and you'll probably use it often.

6.2: Writing Do Loops

Do loops are valuable because occasionally you can't know in advance how many times a loop should repeat. As an alternative to a **For...Next** loop, you can write a **Do** loop that executes statements until a certain condition in the loop is true.

For example, you might want to let the user enter names in a database until the user types *Done* in an input box. In that case, you could use a **Do** loop to cycle indefinitely until the user enters the text string, *Done*.

6.2.1: Anatomy of a Do Loop

A **Do** loop has several formats, which depend on where and how Visual Basic evaluates the loop condition.

A Standard Do Loop

The most common **Do** loop syntax looks like this:

```
Do While condition
    block of statements to be executed
Loop
```

For example, this **Do** loop consists of statements that process input until the user enters the word *Done*:

```
Do While InpName <> "Done"
    InpName = InputBox("Type a name or Done to quit.")
    If InpName <> "Done" Then Print InpName
Loop
```

The conditional statement in this loop is `InpName <> "Done"`. The Visual Basic compiler translates this statement to mean "loop as long as the InpName variable doesn't contain the word `"Done."`

Note This code brings up an interesting fact about **Do** loops: if the condition at the top of the loop is not **True** when the **Do** statement is first evaluated, Visual Basic never executes the **Do** loop. Suppose, that the InpName variable *did* contain the text string "Done" before the loop started (perhaps from an earlier assignment in the event procedure). Visual Basic would skip the loop altogether and continue with the line below the **Loop** keyword.

Also, note that this type of loop requires an extra **If...Then** structure to prevent the exit value from being displayed when the user types it.

Conditional Test at the Bottom of the Loop

If you want the loop to always run at least once in a program, put the conditional test at the bottom of the loop. For example, this loop is essentially the same as the **Do** loop shown above, but the loop condition is tested after a name is received from the **InputBox** function:

```
Do
    InpName = InputBox("Type a name or Done to quit.")
    If InpName <> "Done" Then Print InpName
Loop While InpName <> "Done"
```

The advantage of this syntax is that you can update the InpName variable before the conditional test in the loop. This syntax prevents a preexisting "Done" value causing the loop to be skipped. Testing the loop condition at the bottom ensures that your loop will be executed at least once, but often, you'll need to add a few extra statements to process the data.

6.2.2: Avoiding an Endless Loop

Do loops are relentless, so it is very important that you design test conditions carefully. Each loop must have a **True** exit point. If a loop test never evaluates to **False**, the loop will execute endlessly, and your program will no longer respond to input.

Consider the following example:

```
Do
    Number = InputBox("Enter number to square, or -1 to quit.")
    Number = Number * Number
    Print Number
Loop While Number >= 0
```

In this loop, the user enters number after number, and the program squares each number and prints it on the form. Unfortunately, when the user has had enough, he or she can't quit because the advertised exit condition doesn't work.

When the user enters **-1**, the program squares it, and the Number variable is assigned the value 1. (You can fix this logic error by setting a different exit condition.)

It's a good thing to watch for endless **Do** loops. Fortunately, faulty exit conditions are pretty easy to spot if you test your program thoroughly.

6.2.3: Using the Until Keyword

So far, the **Do** loops you have seen use the **While** keyword to execute statements as long as the loop condition remained **True**. In Visual Basic, you can also use the **Until** keyword in **Do** loops to cycle *until* a certain condition is true.

Testing a Condition

To test a condition, you can use the **Until** keyword at the top or bottom of a **Do** loop, just as you use the **While** keyword. For example, the following **Do** loop uses the **Until** keyword to loop repeatedly until the user enters the word *Done* in an input box:

```
Do
    InpName = InputBox("Type a name or Done to quit.")
    If InpName <> "Done" Then Print InpName
Loop Until InpName = "Done"
```

Test Condition Operators

As you can see, a loop that uses the **Until** keyword is very similar to a loop that uses the While keyword. In our example, the only difference is that the test condition usually contains the equal to operator (**=**) versus the not-equal-to operator **(<>**).

If using the **Until** keyword makes sense to you, feel free to use it with test conditions in your **Do** loops.

6.3: Using the Timer Control

To execute a group of statements for a specified period of time, you can use the **Timer** control in the Visual Basic toolbox. The **Timer** control is an invisible stopwatch that gives your programs access to the system clock. You can use the **Timer** to:

- Count down from a preset time, like an egg timer.
- Delay a program.
- Repeat an action at prescribed intervals.

Objects that you create with the Visual Basic Timer:

- Are accurate to 1 millisecond (1/1000 of a second).
- Aren't visible at run time.
- Are associated with an event procedure that runs whenever the preset timer interval elapses.

Setting the Timer Interval

To set a timer interval, you start by using the timer **Interval** property. Then, you activate the timer by setting the timer **Enabled** property to **True**. When a timer is enabled, it runs constantly. The program executes the timer event procedure at the prescribed interval — until the user stops the program, the timer is disabled, or the **Interval** property is set to 0.

6.3.1: EXAMPLE: Creating a Digital Clock

A digital clock is one of the most practical uses for a timer. In this example we'll create a digital clock using a timer and a simple label object on a form step by step.

Step1: Creating The User Interface

Open a new project and place a timer object on your form. Then locate a label at the center of the form and adjust the size as shown in the figure below. For such a program a better look will be established by sizing your form as a pop up window.

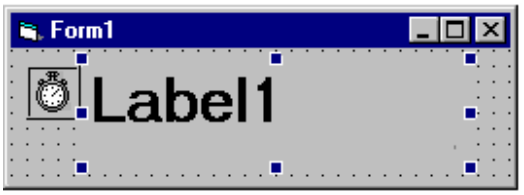


Figure 1: Form with controls

Step2: Setting the Properties

Set the properties of each object as follows,

Form

Caption	Digital Clock

Timer

Interval	1000
Enabled	True

Label

Alignment	2-Center
Caption	" " (Optional)
Font	Arial, 20 pts, Bold

Step3: Adding Code

We'll write our VB code to the Timer's timer event since the timer is the main object that leads the execution.

Then add the code given below to the Timer,

```
Private Sub Timer1_Timer

Label1.Caption = Time

End Sub
```

Step4: Running The Program

As the program is started, we can see that the timer runs the code written before with the interval of every 1000 miliseconds (1 Sec). So it updates the label1.caption with the system time at every second. Your program looks like the one given in the figure below.

Figure 2: Digital Clock Program

In this example of a digital clock, setting the timer **Interval** property to **1000** directs Visual Basic to update the clock time every 1000 milliseconds (once per second).

Note The Windows operating system is a multi-tasking environment, so other programs will also require processing time. Visual Basic might not always get a chance to update the clock each second, but if it gets behind, it will always catch up. To keep track of the time at other intervals (such as once every tenth of a second), simply adjust the number in the **Interval** property.

6.3.2: Setting a Time Limit for Input

There's another interesting way to use a timer: to wait for a set period of time and then to either enable or prohibit an action. This is a little like setting an egg timer in your program — you set the Interval property with the delay you want, and then you start the clock ticking by setting the **Enabled** property to **True**.

This procedure shows you how to set a time limit for input in a program. In the lab exercises at the end of this chapter, you'll adapt these steps to create a password protection program that times out if the user takes more than 15 seconds to enter a password.

▶ **To set a user input time limit**

1. Create a **Timer** object on your form.
2. Set the **Interval** property of the **Timer** object to the user input time limit.

Be sure to specify the time limit in milliseconds. For example, set the **Interval** property to 30000 for a 30-second time limit.

3. In the **Timer** object's Timer event procedure, place statements that print a "time expired" message and that stop the program. For example:

```
MsgBox ("Sorry, your time is up.")
End
```

▶ **To create an event procedure to manage user input**

1. At the place you want to begin the timed input interval, type the following program statement to start the clock:

```
Timer1.Enabled = True
```

You can associate the event procedure with a command button, text box, or any other object that receives input. If you want the clock to start when the form first appears, place the statement in the Form_Load event procedure.

2. To turn off the clock when the user completes the input satisfactorily, use an event procedure with the following statement:

```
Timer1.Enabled = False
```

Without this statement, the timer object event procedure automatically closes the program when the allotted time expires.

-END OF CHAPTER 6-