

Chapter 8: Looping and Counting - Do it Again and Again.

So far our program has started, gone step by step through our instructions, and quit. While this is OK for simple programs, most programs will have tasks that need to be repeated, things counted, or both. This chapter will show you the three looping statements, how to speed up your graphics, and how to slow the program down.

The For Loop:

The most common loop is the **for** loop. The **for** loop repeatedly executes a block of statements a specified number of times, and keeps track of the count. The count can begin at any number, end at any number, and can step by any increment. Program 38 shows a simple **for** statement used to say the numbers 1 to 10 (inclusively). Program 39 will count by 2 starting at zero and ending at 10.

```
1 # for.kbs
2 for t = 1 to 10
3     print t
4     say t
5 next t
```

Program 38: For Statement

```
1
2
3
4
5
6
7
```

```
8
9
10
```


Sample Output 38: For Statement

```
1 # forstep2.kbs
2 for t = 0 to 10 step 2
3     print t
4     say t
5 next t
```

Program 39: For Statement – With Step

```
0
2
4
6
8
10
```

Sample Output 39: For Statement – With Step

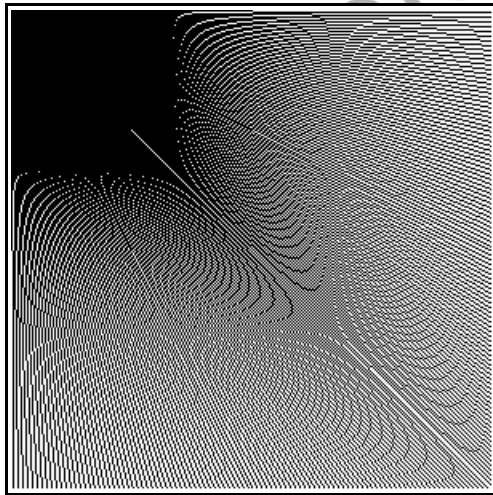
 <p>New Concept</p>	<pre>for variable = expr1 to expr2 [step expr3] statement(s) next variable</pre> <p>Execute a specified block of code a specified number of times. The <i>variable</i> will begin with the value of <i>expr1</i>. The <i>variable</i> will be incremented by <i>expr3</i> (or one if step is not specified) the second and subsequent time through the loop. Loop terminates if <i>variable</i> exceeds <i>expr2</i>.</p>
---	---

Using a loop we can easily draw very interesting graphics. Program 40 will

draw a Moiré Pattern. This really interesting graphic effect is caused by the computer being unable to draw perfectly straight lines. What is actually drawn are pixels in a stair step fashion to approximate a straight line. If you look closely at the lines we have drawn you can see that they actually are jagged.

```
1 # moire.kbs - draw a moire pattern
2
3 clg white
4 color black
5 for t = 1 to 300 step 3
6     line 0,0,300,t
7     line 0,0,t,300
8 next t
```

Program 40: Moiré Pattern



Sample Output 40: Moiré Pattern

**Explore**

What kind of Moiré Patterns can you draw? Start in the center, use different step values, overlay one on top of another, try different colors, go crazy.

For statements can even be used to count backwards. To do this set the **step** to a negative number.

```
1 # stepneg1.kbs
2
3 for t = 10 to 0 step -1
4     print t
5     pause 1.0
6 next t
```

Program 41: For Statement – Countdown

```
10
9
8
7
6
5
4
3
2
1
0
```

Sample Output 41: For Statement – Countdown



`pause seconds`

The **pause** statement tells BASIC-256 to stop executing the current program for a specified number of seconds. The number of seconds may be a decimal number if a fractional second pause is required.

Do Something Until I Tell You To Stop:

The next type of loop is the **do/until**. The **do/until** repeats a block of code one or more times. At the end of each iteration a logical condition is tested. The loop repeats as long as the condition is *false*. Program 42 uses the **do/until** loop to repeat until the user enters a number from 1 to 10.

```

1  # dountil.kbs
2
3  do
4      inputinteger "enter an integer from 1 to 10?",n
5      until n>=1 and n<=10
6      print "you entered " + n

```


Program 42: Get a Number from 1 to 10

```

enter an integer from 1 to 10?66
enter an integer from 1 to 10?-56
enter an integer from 1 to 10?3
you entered 3

```

Sample Output 42: Get a Number from 1 to 10

 <p>New Concept</p>	<pre>do <i>statement(s)</i> until <i>condition</i></pre>
	<p>Do the statements in the block over and over again while the condition is <i>false</i>.</p> <p>The statements will be executed one or more times.</p>

Do Something While I Tell You To Do It:

The third type of loop is the **while/end while**. It tests a condition before executing each iteration and if it evaluates to true then executes the code in the loop. The **while/end while** loop may execute the code inside the loop zero or more times.

Sometimes we will want a program to loop forever, until the user stops the program. This can easily be accomplished using the Boolean *true* constant (see Program 43).


```
1 # whiletrue.kbs
2
3 while true
4     print "nevermore ";
5 end while
```

Program 43: Loop Forever

```
nevermore.
nevermore.
nevermore.
nevermore.
nevermore.
```

... runs until you stop it

Sample Output 43: Loop Forever

	<pre>while condition statement(s) end while</pre>
	<p>Do the statements in the block over and over again while the condition is true.</p> <p>The statements will be executed zero or more times.</p>

Program 44 uses a while loop to count from 1 to 10 like Program 38 did with a **for** statement.

```
1 # whilefor.kbs
2
3 t = 1
4 while t <= 10
5     print t
6     t = t + 1
7 end while
```

Program 44: While Count to 10

```
1
2
3
4
5
6
7
```

```
8
9
10
```

Sample Output 44: While Count to 10

Continuing and Exiting Loops


Sometimes it becomes necessary for a programmer to jump out of a loop before it would normally terminate (exit) or to start the next loop (continue) without executing all of the code.


```
1 # exitwhile.kbs - adding machine
2
3 total = 0
4 while true
5     inputfloat "Enter Value (-999 to exit) > ", v
6     if v = -999 then exit while
7     total = total + v
8 end while
9
10 print "Your total was " + total
```

Program 45: Adding Machine - Using Exit While

```
Enter Value (-999 to exit) > 34
Enter Value (-999 to exit) > -34
Enter Value (-999 to exit) > 234
Enter Value (-999 to exit) > 44
Enter Value (-999 to exit) > -999
Your total was 278.0
```

Sample Output 45: Adding Machine - Using Exit While

 New Concept	<pre>exit do exit for exit while</pre> <p>Jump out of the current loop and skip the remaining code in the loop.</p>
---	---

 New Concept	<pre>continue do continue for continue while</pre> <p>Do not execute the rest of the code in this loop but loop again like normal.</p>
---	--

Fast Graphics:

When we need to execute many graphics quickly, like with animations or games, BASIC-256 offers us a fast graphics system. To turn on this mode you execute the **fastgraphics** statement. Once **fastgraphics** mode is started the graphics output will only be updated once you execute the **refresh** statement.



New Concept

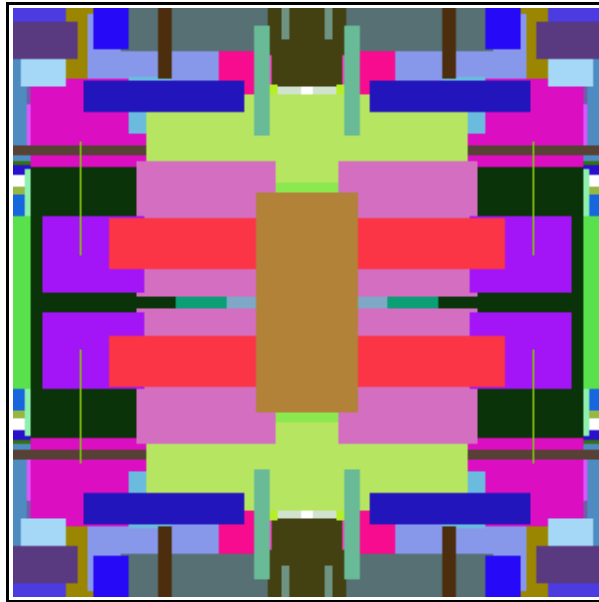
fastgraphics
refresh

Start the **fastgraphics** mode. In fast graphics the screen will only be updated when the **refresh** statement is executed.

Once a program executes the **fastgraphics** statement it can not return to the standard graphics (slow) mode.

```
1  # kaleidoscope.kbs
2
3  clg
4  fastgraphics
5  while true
6      for t = 1 to 100
7          r = int(rand * 256)
8          g = int(rand * 256)
9          b = int(rand * 256)
10         x = int(rand * 300)
11         y = int(rand * 300)
12         h = int(rand * 100)
13         w = int(rand * 100)
14         color rgb(r,g,b)
15         rect x,y,w,h
16         rect 300-x-w,y,w,h
17         rect x,300-y-h,w,h
18         rect 300-x-w,300-y-h,w,h
19     next t
20     refresh
21     pause 1
22 end while
```

Program 46: Kaleidoscope



Sample Output 46: Kaleidoscope



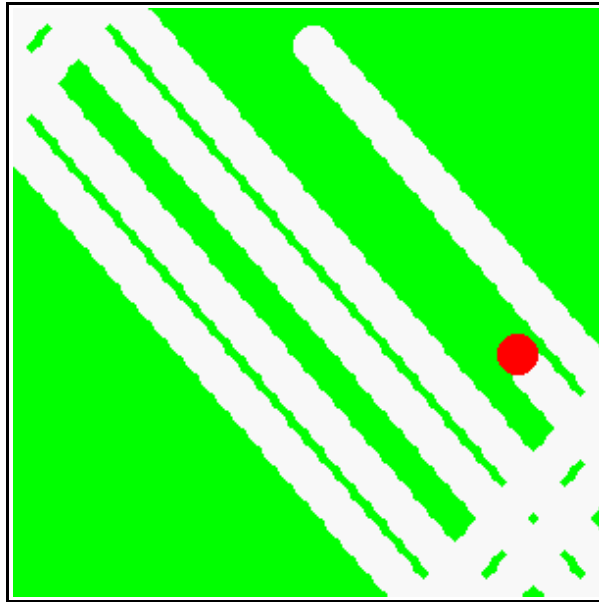
Big Program

In this chapter's "Big Program" let's use a while loop to animate a ball bouncing around on the graphics display area.

```
1 # bouncingball.kbs
2
3 fastgraphics
4
5 # starting position of ball
6 x = rand * 300
7 y = rand * 300
8 # size of ball
```

```
9      r = 10
10     # speed in x and y directions
11     dx = rand * r - r / 2
12     dy = rand * r - r / 2
13
14     clg green
15
16     while true
17         # erase old ball
18         color white
19         circle x,y,r
20         # calculate new position
21         x = x + dx
22         y = y + dy
23         # if off the edges turn the ball around
24         if x < 0 or x > 300 then
25             dx = dx * -1
26             sound 1000,50
27         end if
28         # if off the top or bottom turn the ball around
29         if y < 0 or y > 300 then
30             dy = dy * -1
31             sound 1500,50
32         end if
33         # draw new ball
34         color red
35         circle x,y,r
36         # update the display
37         refresh
38         # slow the ball down
39         pause .05
40     end while
```


Program 47: Big Program - Bouncing Ball




Sample Output 47: Big Program - Bouncing Ball

Free eBook

Exercises:

 <p>Word Search</p>	<pre> f l g b w p e t s w i i f a w t b q l i t n u i t n s n v h p h b c f e i a k t c v r o o e l l x d r k g e w n o i l c e x o u f r d e h l o i i g f r y i a w l n l c t x e n t g d p t i w k g s d i o n e i h p h a h w o a e d n z m i g w x n s d z u u d w t c d x o m i e h d g m o v s </pre> <p>condition, continue, do, endwhile, exit, fastgraphics, for, loop, next, refresh, step, until, while</p>
---	---

 <p>Problems</p>	<ol style="list-style-type: none"> 1. Write a program that uses the for loop to sum the integers from 1 to 42 and display the answer. Hint: before the loop assign a variable to zero to accumulate the total. 2. Write a program that asks the user for an integer from 2 to 12 in a loop. Keep looping until the user enters a number in the range. Calculate the factorial (n!) of the number using a for loop and display it. Remember 2! is 1*2, 3! is 1*2*3, and n! is n * (n-1)!. 3. Write a program to display one through 8 multiplied by 1 through 8. Hint: use a for loop inside another for loop. Format your output to look like:
--	---

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
...
```

4. Re-write #3 to make your output in table format, like:

1	2	3	4	5	6	7	8
2	4	6	8	10	12	14	16
3	6	9	12	15	18	21	24
4	8	12	16	20	24	28	32
5	10	15	20	25	30	35	40
6	12	18	24	30	36	42	48
7	14	21	28	35	42	49	56
8	16	24	32	40	48	56	64