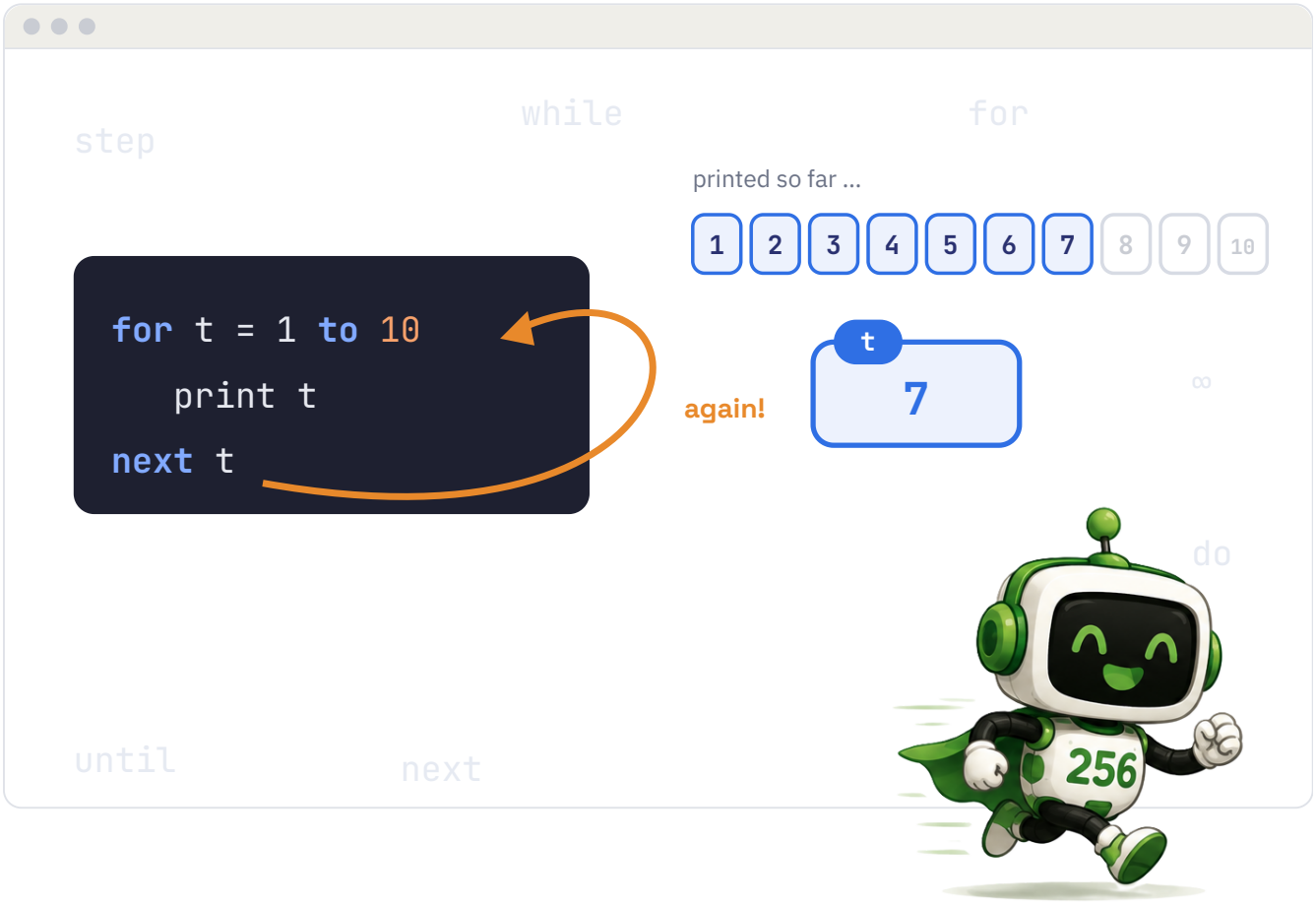
LEARN TO CODE · LOOPS

Chapter 8

Looping and Counting — Do It Again and Again

So far, every program has started, gone step by step through its statements, and stopped. Most real programs have work that must be repeated, things to count, or both. This chapter shows you the three kinds of loop, how to slow a program down, and how to make your graphics fast enough for animation.



© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 1

The for loop

The `for` loop repeats a block of statements a set number of times, and counts as it goes.

The most common loop is the **for** loop. It uses a variable to count: you say where the count starts, where it ends and, if you like, how big each step is. Program 43 prints and says the numbers 1 to 10.

for.kbs — BASIC-256

```
# for.kbs - count from 1 to 10
for t = 1 to 10
   print t
   say t
next t
```

TEXT OUTPUT

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

says each number

Program 43 · Count to Ten

Add **step** to count in bigger steps. Program 44 counts by 2, starting at 0 and ending at 10.

forstep2.kbs — BASIC-256

```
# forstep2.kbs - count by 2
for t = 0 to 10 step 2
   print t
   say t
next t
```

TEXT OUTPUT

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

says each number

Program 44 · Count by Two

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



NEW CONCEPT

```
for variable = start to end [step increment]
   statements
next variable
```

The **for** loop sets *variable* to *start* and runs the statements. At **next** it adds *increment* to the variable (1 if there is no **step**) and goes round again, until the variable has passed *end*. With a negative step it counts down and stops once the variable is below *end*.

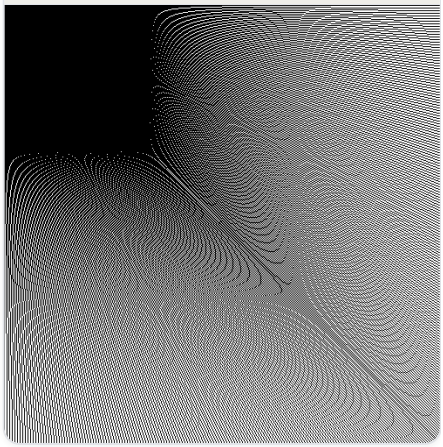
If the variable is already past the end at the start — `for t = 1 to 0` — the statements are **not run at all**. After `next` you may leave out the variable name, but writing it makes long programs easier to read.

With a loop, a few lines can draw a lot. Program 45 draws a fan of lines from the top left corner to the right and bottom edges, 3 pixels apart.

moire.kbs BASIC-256

```
# moire.kbs - draw a moire pattern
clg white
color black
for t = 1 to 500 step 3
   line 0,0,500,t
   line 0,0,t,500
next t
```

GRAPHICS OUTPUT



Program 45 · Moiré Pattern

Where do the curves come from? The program only draws straight lines! The graphics output is a grid of tiny squares called **pixels**, so a sloping line has to be drawn as a staircase of pixels. Where many lines lie close together, their staircases line up into curved bands. This effect is called a **moiré pattern**.



EXPLORE

What moiré patterns can you make? Start from the centre of the screen, try different step values, draw one fan on top of another, use different colours — go crazy.

A **for** loop can also count backwards: give it a negative step. Program 46 counts down from 10, one number a second.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

countdown.kbs BASIC-256

```
# countdown.kbs - count backwards, one number a second
for t = 10 to 0 step -1
   print t
   pause 1
next t
print "Lift off!"
```

Program 46 · Countdown

TEXT OUTPUT

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



NEW CONCEPT

```
pause seconds
```

The **pause** statement makes the program wait for a number of seconds before going on. The number may have a decimal point: `pause 0.25` waits a quarter of a second.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 2

Do something until I tell you to stop

The `do` ... `until` loop repeats a block until a condition becomes true.

The second kind of loop is **do** ... **until**. It runs the block first, and checks its condition at the end. If the condition is *false*, it goes round again; once it is *true*, the loop is finished. Program 47 keeps asking until the user types a number from 1 to 10.

dountil.kbs — BASIC-256

```
# dountil.kbs - ask until the answer is right
do
   inputinteger "Enter a whole number from 1 to 10: ", n
until n >= 1 and n <= 10
print "You entered " + n
```

TEXT OUTPUT

```
Enter a whole number from 1 to 10: 66
Enter a whole number from 1 to 10: -56
Enter a whole number from 1 to 10: 3
You entered 3
```

Program 47 · Get a Number from 1 to 10



NEW CONCEPT

```
do
   statements
until condition
```

The statements between **do** and **until** are repeated as long as the condition is *false*. Because the condition is checked at the end, the statements always run **at least once** — just right for asking a question, which you always have to ask at least once.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 3

Do something while I tell you to do it

The `while` loop checks its condition first, and repeats as long as it is true.

The third kind of loop is **while** … **end while**. It checks the condition *before* each time round. If the condition is true, it runs the block; if it is false — even the very first time — the loop is skipped.

Sometimes you want a program to loop forever, until the user stops it. Use the Boolean constant **true** from Chapter 7 as the condition: it is never false.

whiletrue.kbs — BASIC-256

```
# whiletrue.kbs - loop forever
while true
   print "nevermore ";
end while
```

TEXT OUTPUT

```
nevermore nevermore nevermore nevermore nevermore
nevermore nevermore nevermore nevermore nevermore
nevermore nevermore nevermore nevermore nevermore
nevermore nevermore nevermore nevermore nevermore
nevermore nevermore nevermore
```

… and on, and on — until you press **Stop**

Program 48 · Loop Forever

The semicolon keeps every “nevermore” on the same line; the text output wraps it at the edge of the window. To end a program that never stops by itself, click **Stop** on the toolbar, choose **Run ▸ Stop** or press **Shift+F5**.



NEW CONCEPT

```
while condition
   statements
end while
```

The statements between **while** and **end while** are repeated as long as the condition is *true*. The condition is checked before each time round, so the statements may run **zero or more times**. You may write **end while** or **endwhile**.

A **while** loop can count, too. Program 49 counts from 1 to 10 like Program 43 — but now you have to do the counting yourself: set the variable before the loop and add 1 to it with `+=` inside the loop.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

whilefor.kbs BASIC-256

```
# whilefor.kbs - count to 10 with while
t = 1
while t <= 10
   print t
   t += 1
end while
```

TEXT OUTPUT

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

Program 49 · Count to Ten with While

Forgot to count? Leave out `t += 1` and `t` stays 1 forever: Program 49 prints 1 until you press Stop. When a loop never ends, check that something inside it changes the condition.

Which loop should I use?

for

When you know how many times, or you need to count.

Checks before each time round: may run zero times.

do … until

When the block must run at least once, like asking a question.

Checks at the end: runs one or more times.

while

When you repeat as long as something is true — or forever.

Checks before each time round: may run zero times.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 4

Leaving a loop early, and skipping ahead

With `exit` you jump out of a loop; with `continue` you skip straight to the next time round.

Sometimes you need to leave a loop before it would normally finish. Program 50 is an adding machine: it adds up the numbers you type until you type −999. The loop itself would go on forever; the **exit while** statement jumps out of it.

exitwhile.kbs — BASIC-256

```
# exitwhile.kbs - adding machine
total = 0
while true
   inputfloat "Enter a value (-999 to stop): ", v
   if v = -999 then exit while
   total += v
end while
print "Your total was " + total
```

TEXT OUTPUT

```
Enter a value (-999 to stop): 34
Enter a value (-999 to stop): -34
Enter a value (-999 to stop): 234
Enter a value (-999 to stop): 44
Enter a value (-999 to stop): -999
Your total was 278.0
```

Program 50 · Adding Machine — Using Exit While



NEW CONCEPT

```
exit do
exit for
exit while
```

exit jumps out of the loop straight away: the rest of the block is skipped, and the program carries on after the end of the loop. Use the word for the kind of loop you are in.

continue skips the rest of the block too, but instead of leaving the loop it goes straight round again. Program 51 refuses negative numbers: it prints a message and continues with the next question, so the number is never added.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

continue.kbs BASIC-256

```
# continue.kbs - adding machine for positive numbers
total = 0
while true
   inputfloat "Enter a value (-999 to stop): ", v
   if v = -999 then exit while
   if v < 0 then
      print "Positive numbers only, please."
      continue while
   end if
   total += v
end while
print "Your total was " + total
```

TEXT OUTPUT

```
Enter a value (-999 to stop): 34
Enter a value (-999 to stop): -34
Positive numbers only, please.
Enter a value (-999 to stop): 234
Enter a value (-999 to stop): 44
Enter a value (-999 to stop): -999
Your total was 312.0
```

Program 51 · Adding Machine — Using Continue While



NEW CONCEPT

```
continue do
continue for
continue while
```

continue skips the rest of the block and starts the next time round the loop. In a **for** loop the variable is still counted on; in a **do** or **while** loop the condition is checked as usual.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 5

Fast graphics

For animations and games, draw a whole picture first and then show it all at once.

Normally every circle, line and rectangle appears on the screen as soon as it is drawn. That is easy to follow, but slow, and a moving picture flickers. With **fastgraphics** switched on, BASIC-256 draws out of sight, and the graphics output only changes when you say **refresh**.



NEW CONCEPT

```
fastgraphics
refresh
```

fastgraphics switches fast graphics on. From then on the graphics output is only updated when the program executes **refresh**. Fast graphics stay on until the program stops.

Program 52 draws 100 rectangles in random colours, each one four times, mirrored left to right and top to bottom. Then it shows them all at once, waits a second, and draws 100 more on top — forever. You met `rgb` in Chapter 2.



TRY IT

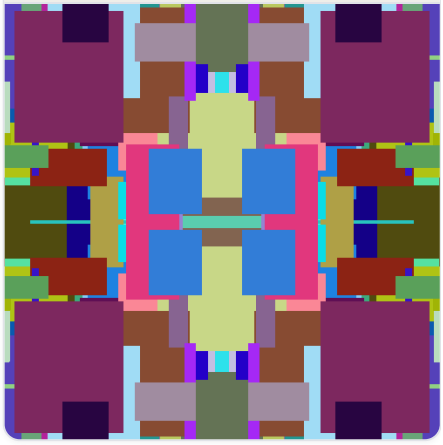
Run Program 52, then put a `#` in front of `fastgraphics`, so that BASIC-256 ignores it, and run it again. Now you can watch every rectangle being drawn, one after the other.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

kaleidoscope.kbs BASIC-256

```
# kaleidoscope.kbs - mirrored random rectangles
clg black
fastgraphics
while true
   for t = 1 to 100
      r = int(rand * 256)
      g = int(rand * 256)
      b = int(rand * 256)
      x = int(rand * 500)
      y = int(rand * 500)
      w = int(rand * 160)
      h = int(rand * 160)
      color rgb(r,g,b)
      rect x,y,w,h
      rect 500-x-w,y,w,h
      rect x,500-y-h,w,h
      rect 500-x-w,500-y-h,w,h
   next t
   refresh
   pause 1
end while
```

GRAPHICS OUTPUT



After the first refresh.

Program 52 · Kaleidoscope

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



BIG PROGRAM

Bouncing Ball

This chapter's big program uses a **while** loop and fast graphics to bounce a ball around the graphics output — with a beep every time it hits a wall.

bouncingball.kbs BASIC-256

```
# bouncingball.kbs - a ball bouncing around the graphics output
fastgraphics

# size of the ball
r = 15
# starting position of the ball
x = r + rand * (500 - 2 * r)
y = r + rand * (500 - 2 * r)
# speed in the x and y directions
dx = 2 + rand * 4
dy = 2 + rand * 4
if rand < 0.5 then dx = -dx
if rand < 0.5 then dy = -dy

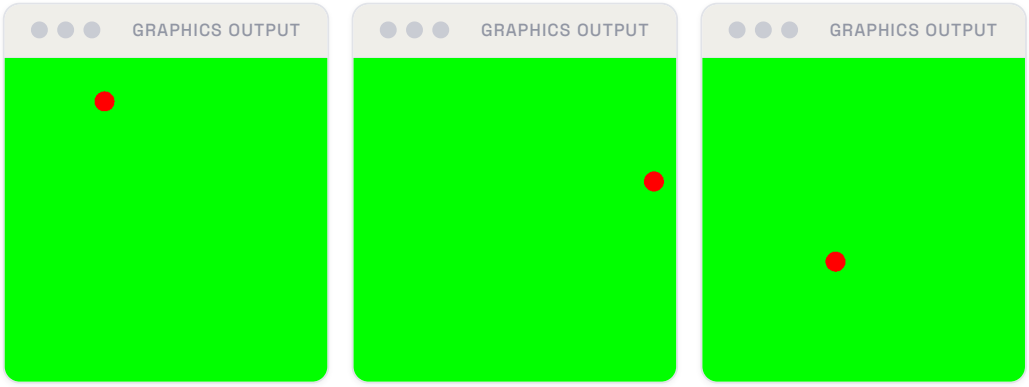
while true
   # clear the screen and draw the ball
   clg green
   color red
   circle x,y,r
   # show the new picture
   refresh
   # calculate the new position
   x += dx
   y += dy
   # past the left or right edge? turn around
   if x < r or x > 500 - r then
      dx = -dx
      soundplay 1000,50
   end if
   # past the top or bottom? turn around
   if y < r or y > 500 - r then
      dy = -dy
      soundplay 1500,50
   end if
   # slow the ball down
   pause 0.02
end while
```

Program 53 · Big Program — Bouncing Ball

Each time round the loop is one frame of the animation: clear the screen, draw the ball, show the picture, move the ball. `dx` and `dy` are how far the ball moves across and down in each frame; at least 2 pixels, so it never crawls, and with a random direction. When the ball reaches an edge, the minus sign turns its

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

direction around. The edges are tested with `r`, the radius, so that the ball bounces when its side touches the wall, not its centre.



Three frames from one run — on screen the ball moves smoothly.



NEW CONCEPT

```
soundplay frequency, duration
```

soundplay plays tones just like **sound** from Chapter 4, but it does **not wait** for the sound to finish: the program carries on at once. With **sound** the ball would stop for a moment at every bounce. You will learn more about **soundplay** in the **Advanced Sound and Music** chapter.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



PRACTICE

Exercises

WORD SEARCH

a c p o s i h s e r f e r
e t d l o l i t n u y l e
l w h p u s k a e l t h n
i a u o n t r d l v c r o
h v u o d x o o i f u e i
w u h l p e f y h m w x t
d e e r l n p v w y l i i
n s c u a p k a h r r t d
e u g h y p f p l l f p n
u a d c o n t i n u e t o
v p b r i n m y s t j a c
i u u u c g p h s g e n p
b s c i h p a r g t s a f

Find: condition, continue, do, endwhile, exit, fastgraphics, for, loop, next, pause, refresh, soundplay, step, until, while



Problems

Try these — they only use what this chapter and the earlier ones covered.

1 Write a program that uses a **for** loop to add up the whole numbers from 1 to 42 and print the answer. Hint: before the loop, set a variable to 0; inside the loop, add the counter to it with `+=`.

2 Write a program that asks the user for a whole number from 2 to 12, and keeps asking until the number is in range. Then calculate its factorial (written *n*!) with a **for** loop and print it. Remember: 2! is 1×2, 3! is 1×2×3, and *n*! is *n* × (*n*−1)!.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

3 Write a program that prints 1 to 8 multiplied by 1 to 8. Hint: put a **for** loop inside another **for** loop. Make your output look like this:

```
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 Rewrite your program from problem 3 so that it prints a table. Hint: print each number with a semicolon, so the row stays on one line, and use **if** to print an extra space in front of the numbers below 10. After each row, an empty `print` starts the next line.

```
 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
```

5 Draw a target: use a **for** loop with a negative step to draw circles in the middle of the graphics output, from radius 250 down to 25, changing colour between red and white each time. Hint: a variable that keeps switching between 0 and 1, with `c = 1 - c`.

6 Change Program 53 so that the ball gets a new random colour every time it bounces. You need three new variables for the red, green and blue parts (Chapter 2's **rgb**) — but careful, **r** is already taken!

Chapter 8 · Looping and Counting — Do It Again and Again. Modernised layout of the BASIC-256 beginner's guide, updated for BASIC-256 2.3. Programs are numbered 43–53, continuing from Chapter 7. New: Program 51 shows `continue`, which the original only described; a table of which loop to use; a **for** loop can run zero times; how to stop a program. Corrected: the Loop Forever output (the semicolon keeps the words on one line), the explanation of the moiré pattern, and the Big Program, which erased the ball with white on a green background and so left white trails. It now clears each frame, bounces at the ball's edge, never moves too slowly, and beeps with `soundplay` so the animation doesn't stop. Programs are redrawn for the 500×500 graphics output and use +=. Every sample output was produced by running the program in BASIC-256 2.3. Original text © 2019 James M. Reneau, licensed under CC BY-NC-SA 3.0 US. This adaptation is shared under the same licence. Mascot: BitBot.