Chapter 9: Functions and Subroutines – Reusing Code.

This chapter introduces the use of Functions and Subroutines. Programmers create subroutines and functions to test small parts of a program, reuse these parts where they are needed, extend the programming language, and simplify programs.

Functions:

A function is a small program within your larger program that does something for you. You may send zero or more values to a function and the function will return one value. You are already familiar with several built in functions like: `rand` and `rgb`. Now we will create our own.

![Illustration 22: Block Diagram of a Function](image_url)
The `Function` statement creates a new named block of programming statements and assigns a unique name to that block of code. It is recommended that you do not name your function the same name as a variable in your program, as it may cause confusion later.

In the required parenthesis you may also define a list of variables that will receive values from the “calling” part of the program. These variables belong to the function and are not available to the part of the program that calls the function.

A function definition must be closed or finished with an `End Function`. This tells the computer that we are done defining the function.

The value being returned by the function may be set in one of two ways: 1) by using the `return` statement with a value following it or 2) by setting the function name to a value within the function.

### Return value

Execute the `return` statement within a function to return a value and send control back to where it was called from.
Terminates the program (stop).

Program 56: Minimum Function

```
# c9_minimum.kbs
# minimum function

input "enter a number ", a
input "enter a second number ", b

print "the smaller one is ";
print minimum(a,b)
end

function minimum(x,y)
    # return the smallest of the two numbers passed
    if x<y then return x
    return y
end function
```

Sample Output 56: Minimum Function

```
enter a number 7
enter a second number 3
the smaller one is 3
```
Chapter 9: Functions and Subroutines – Reusing Code.

Program 57: Game Dice Roller

die roller
sides on the die (default 6) ?6
number of die (default 2) ?3
6
3
1
total 10

© 2014 James M. Reneau (CC BY-NC-SA 3.0 US)
Sample Output 57: Game Dice Roller

In the examples above we have created functions that returned a numeric value. Functions may also be created that return a string value. A string function, like a variable, has a dollar sign after its name to specify that it returns a string.

Program 58: Simple String Function

```
# c9_repeatstring.kbs
# simple string function - make copies

a$ = "hi"
b$ = repeat$(a$,20)
print a$
print b$
end

function repeat$(w$,n)
a$ = ""
for t = 1 to n
    a$ += w$
next t
return a$
end function
```

Sample Output 58: Simple String Function

```
hi
hihihihihihihihihihihihihihihihihihihihihihi
```

Observe in the function samples, above, that variables within a function exist only within the function. If the same variable name is used in the function it DOES NOT change the value outside the function.
Subroutines:

A subroutine is a small subprogram within your larger program that does something specific. Subroutines allow for a single block of code to be used by different parts of a larger program. A subroutine may have values sent to it to tell the subroutine how to react.

Subroutines are like functions except that they do not return a value and that they require the use of the call statement to execute them.

```
Subroutine subroutinename( argument(s) )
    statements
End Subroutine
```

The `Subroutine` statement creates a new named block of programming statements and assigns a unique name to that block of code. It is recommended that you do not name your subroutine the same name as a variable in your program, as it may cause confusion later.

In the required parenthesis you may also define a list of variables that will receive values from the “calling” part of the program. These variables are local to the subroutine and are not directly available to the calling program.

A subroutine definition must be closed or finished with an `End Subroutine`. This tells the computer that we are done defining the subroutine.
Call subroutineName( value(s))

The Call statement tells BASIC-256 to transfer program control to the subroutine and pass the values to the subroutine for processing.

Return

Execute the return statement within a subroutine to send control back to where it was called from.

This version of the return statement does not include a value to return, as a subroutine does not return a value.

```kbs
# c9_subroutineclock.kbs
# display a comple ticking clock

fastgraphics
font "Tahoma", 20, 100
color blue
rect 0, 0, 300, 300
color yellow
text 0, 0, "My Clock."

while true
  call displaytime()
  pause 1.0
end while
end
```

© 2014 James M. Reneau (CC BY-NC-SA 3.0 US)
subroutine displaytime()
  color blue
  rect 100, 100, 200, 100
  color yellow
  text 100, 100, hour + "::" + minute + "::" + second
  refresh
end subroutine

Program 59: Subroutine Clock

Sample Output 59: Subroutine Clock
The functions `year`, `month`, `day`, `hour`, `minute`, and `second` return the components of the system clock. They allow your program to tell what time it is.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>year</code></td>
<td>Returns the system 4 digit year.</td>
</tr>
<tr>
<td><code>month</code></td>
<td>Returns month number 0 to 11. 0 – January, 1-February...</td>
</tr>
<tr>
<td><code>day</code></td>
<td>Returns the day of the month 1 to 28,29,30, or 31.</td>
</tr>
<tr>
<td><code>hour</code></td>
<td>Returns the hour 0 to 23 in 24 hour format. 0 – 12 AM, 1-1 AM,... 12 – 12 PM, 13 – 1 PM, 23 – 11 PM...</td>
</tr>
<tr>
<td><code>minute</code></td>
<td>Returns the minute 0 to 59 in the current hour.</td>
</tr>
<tr>
<td><code>second</code></td>
<td>Returns the second 0 to 59 in the current minute.</td>
</tr>
</tbody>
</table>

```plaintext
1  # c9_subroutineclockimproved.kbs
2  # better ticking clock
3
4  fastgraphics
5  font "Tahoma", 20, 100
6  color blue
7  rect 0, 0, 300, 300
8
9  call displayyear()
10  while true
11      call displaytime()
12      pause 1.0
```
end while
end
subroutine displayyear()
color blue
rect 50,50, 200, 100
color yellow
text 50,50, padnumber$(month) + "/" + padnumber$(day) + "/" + padnumber$(year)
refresh
end subroutine

subroutine displaytime()
color blue
rect 50,100, 200, 100
color yellow
text 50, 100, padnumber$(hour) + ":" + padnumber$(minute) + ":" + padnumber$(second)
refresh
end subroutine

function padnumber$(n)
    padnumber$ = string(n)
    if n < 10 then
        padnumber$ = "0" + padnumber$
    end if
end function

Program 60: Subroutine Clock - Improved
Using the Same Code in Multiple Programs:

Once a programmer creates a subroutine or function they may want to re-use these blocks of code in other programs. You may copy and paste the code from one program to another but what if you want to make small changes and want the change made to all of your programs. This is where the include statement comes in handy.

The include statement tells BASIC-256 at compile time (when you first press the run button) to bring in code from other files. In Program 61 (below) you can see that the functions have been saved out as their own files and included back into the main program.

```basic
# c9_gamerollerinclude.kbs
# Game Dice Roller
include "e2_c9_diefunction.kbs"
include "e2_c9_inputnumberfunction.kbs"

print "die roller"
s = inputnumber("sides on the die",6)
n = inputnumber("number of die", 2)
total = 0
for x = 1 to n
d = die(s)
print d
```

Sample Output: 60: Subroutine Clock - Improved
Chapter 9: Functions and Subroutines – Reusing Code.

Now that we have split out the functions we can use them in different programs, without having to change the function code or re-typing it.
include "e2_c9_inputnumberfunction.kbs"

print "adding machine"
print "press stop to end"

total = 0
while true
  a = inputnumber("+ ",0)
  total = total + a
  print total
end while

Program 64: Adding Machine – Using the inputnumber Function

adding machine
press stop to end
+ (default 0) ?6
  6
+ (default 0) ?
  6
+ (default 0) ?55
  61
+ (default 0) ?

Sample Output 64: Adding Machine – Using the inputnumber Function

include “string constant”

Include code from an external file at compile (when run is clicked).

The file name must be in quotes and can not be a variable or other expression.
Labels, Goto, and Gosub:

This section contains a discussion of labels and how to cause your program to jump to them. These methods are how we used to do it before subroutines and functions were added to the language. *These statements can be used to create ugly and overly complex programs and should be avoided.*

In Program 41 Loop Forever we saw an example of looping forever. This can also be done using a label and a `goto` statement.

```plaintext
# c9_goto.kbs

top:
print "hi"
goto top
```

*Program 65: Goto With a Label*

```plaintext
hi
hi
hi
hi
... repeats forever
```

*Sample Output 65: Goto With a Label*
A label allows you to name a place in your program so you may jump to that location later in the program. You may have multiple labels in a single program, but each label can only exist in one place.

A label name is followed with a colon (:); must be on a line with no other statements; must begin with a letter; may contain letters and numbers; and are case sensitive. Also, you can not use words reserved by the BASIC-256 language when naming labels (see Appendix I), or the names of subroutines and functions.

Examples of valid labels include: top:, far999:, and About:.

The `goto` statement causes the execution to jump to the statement directly following the label.

Subroutines and functions allow us to reuse blocks of code. The `gosub` statement also allows a programmer to reuse code. Variables in a `gosub` block are global to the entire program.

Program 66 shows an example of a subroutine that is called three times.

```
1  # c9_gosub.kbs
2  # a simple gosub
```
Chapter 9: Functions and Subroutines – Reusing Code.

4 a = 10
5 for t = 1 to 3
6   print "a equals " + a
7   gosub showline
8 next t
9 end
10 showline:
11 print "------------------"
12 a = a * 2
13 return

Program 66: Gosub

a equals 10
------------------
a equals 20
------------------
a equals 40
------------------

Sample Output 66: Gosub

The **gosub** statement causes the execution to jump to the subroutine defined by the **label**.
In our "Big Program" this chapter, let's make a program to roll two dice, draw them on the screen, and give the total. Let's use an included function to generate the random number of spots and a subroutine to draw the image so that we only have to write it once.

```kbs
# roll two dice graphically
include "e2_c9_diefunction.kbs"

clg

total = 0

roll = die(6)
total = total + roll
call drawdie(30,30, roll)

roll = die(6)
total = total + roll
call drawdie(130,130, roll)

print "you rolled " + total + "."
end

subroutine drawdie(x,y,n)
  # set x,y for top left and n for number of dots
  # draw 70x70 with dots 10x10 pixels
  color black
  rect x,y,70,70
  color white
  # top row
  if n <> 1 then rect x + 10, y + 10, 10, 10
  if n = 6 then rect x + 30, y + 10, 10, 10
```

© 2014 James M. Reneau (CC BY-NC-SA 3.0 US)
if \( n \geq 4 \) and \( n \leq 6 \) then rect \( x + 50, y + 10, 10, 10 \)

# middle

if \( n = 1 \) or \( n = 3 \) or \( n = 5 \) then rect \( x + 30, y + 30, 10, 10 \)

# bottom row

if \( n \geq 4 \) and \( n \leq 6 \) then rect \( x + 10, y + 50, 10, 10 \)

if \( n = 6 \) then rect \( x + 30, y + 50, 10, 10 \)

if \( n \neq 1 \) then rect \( x + 50, y + 50, 10, 10 \)

end subroutine

Program 67: Big Program - Roll Two Dice Graphically

Sample Output 67: Big Program - Roll Two Dice Graphically
Chapter 9: Functions and Subroutines – Reusing Code.

Exercises:

9.1. Create a subroutine that will accept two numbers representing a point on the screen. Have the routine draw a
smiling face with a radius of 20 pixels at that point. You may use circles, rectangles, or polygons as needed. Call that subroutine in a loop 100 times and draw the smiling faces at random locations to fill the screen.

9.2. Write a program that asks for two points \( x_1, y_1 \) and \( x_2, y_2 \) and displays the formula for the line connecting those two points in slope-intercept format \( y = mx + b \). Create a function that returns the slope \( m \) of the connecting line using the formula \( \frac{y_1 - y_2}{x_1 - x_2} \). Create a second function that returns the y intercept \( b \) when the \( x \) and \( y \) coordinates of one of the points and the slope are passed to the function.

\[
x_1? 1 \\
y_1? 1 \\
x_2? 3 \\
y_2? 2 \\
y = 0.5x + 0.5
\]

9.3. In mathematics the term factorial means the product of consecutive numbers and is represented by the exclamation point. The symbol \( n! \) means \( n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 \) where \( n \) is an integer and \( 0! \) is 1 by definition.
Write a function that accepts one number and returns its factorial. Call that new function within a for loop to display 1! to 10!. Your output should look like:

1! is 1
2! is 2
3! is 6
4! is 24
5! is 120
6! is 720
7! is 5040
8! is 40320
9! is 362880
10! is 3628800

9.4. A recursive function is a special type of function that calls itself. Knowing that \( n! = n \times (n-1)! \) and that \( 0! = 1 \) rewrite #9.3 to use a recursive function to calculate a factorial.

© 2014 James M. Reneau (CC BY-NC-SA 3.0 US)