THE BOOK OF™ RUBY

A HANDS-ON GUIDE FOR THE *ADVENTUROUS*

HUW COLLINGBOURNE

no starch press

THE BOOK OF™ RUBY

A Hands-On Guide for the Adventurous

by Huw Collingbourne

no starch press

San Francisco

BRIEF CONTENTS

10

BLOCKS, PROCS, AND LAMBDAS

When programmers talk about blocks, they often mean some arbitrary "chunks" of code. In Ruby, however, a block is special. It is a unit of code that works somewhat like a method but, unlike a method, it has no name.

Blocks are very important in Ruby, but they can be difficult to understand. In addition, there are some important differences in the behavior of blocks in Ruby 1.8 and Ruby 1.9. If you fail to appreciate those differences, your programs may behave in unexpected ways when run in different versions of Ruby. This chapter looks at blocks in great detail. Not only does it explain how they work and why they are special, but it also provides guidance on ensuring that they continue to work consistently no matter which version of Ruby you happen to be using.

What Is a Block?

Consider this code:

1blocks.rb

```
3.times do |i|
    puts( i )
end
```

It's probably pretty obvious that this code is intended to execute three times. What may be less obvious is the value that `i` will have on each successive turn through the loop. In fact, the values of `i` in this case will be 0, 1, and 2. The following is an alternative form of the previous code. This time, the block is delimited by curly brackets rather than by `do` and `end`.

```
3.times { |i|
    puts( i )
}
```

According to the Ruby documentation, `times` is a method of Integer (let's call the Integer `int`), which iterates a block "`int` times, passing in values from 0 to `int` -1." So here, the code within the block is run three times. The first time it is run, the variable `i` is given the value 0; each subsequent time, `i` is incremented by 1 until the final value, 2 (that is, `int`-1), is reached.

The two code examples shown earlier are functionally identical. A block can be enclosed either by curly brackets or by the `do` and `end` keywords, and the programmer can use either syntax according to personal preference.

NOTE *Some Ruby programmers like to delimit blocks with curly brackets when the entire code of the block fits onto a single line and with `do..end` when the block spans multiple lines. My personal prejudice is to be consistent, irrespective of code layout, so I generally use curly brackets when delimiting blocks. Usually your choice of delimiters makes no difference to the behavior of the code—but see "Precedence Rules" on page 170.*

If you are familiar with a C-like language such as C# or Java, you may, perhaps, assume that Ruby's curly brackets can be used, as in those languages, simply to group together arbitrary "blocks" of code—for example, a block of code to be executed when a condition evaluates to true. This is not the case. In Ruby, a block is a special construct that can be used only in very specific circumstances.

Line Breaks Are Significant

You must place the opening block delimiter on the same line as the method with which it is associated.

For example, these are okay:

```
3.times do |i|
    puts( i )
end
```

```
3.times { |i|
    puts( i )
}
```

But these contain syntax errors:

```
3.times
do |i|
    puts( i )
end

3.times
{ |i|
    puts( i )
}
```

Nameless Functions

A Ruby block may be regarded as a sort of nameless function or method, and its most frequent use is to provide a means of iterating over items from a list or range of values. If you have never come across nameless functions, this may sound like gobbledygook. With luck, by the end of this chapter, things will have become a little clearer. Let's look back at the simple example given earlier. I said a block is like a nameless function. Take this block as an example:

```
{ |i|
    puts( i )
}
```

If that were written as a normal Ruby method, it would look something like this:

```
def aMethod( i )
   puts( i )
end
```

To call that method three times and pass values from 0 to 2, you might write this:

```
for i in 0..2
   aMethod( i )
end
```

When you create a nameless method (that is, a block), variables declared between upright bars such as `|i|` can be treated like the arguments to a named method. I will refer to these variables as *block parameters*.

Look again at my earlier example:

```
3.times { |i|
    puts( i )
}
```

The `times` method of an integer passes values to a block from 0 to the specified integer value minus 1.

So, this:

```
3.times{ |i| }
```

is very much like this:

```
for i in 0..2
    aMethod( i )
end
```

The chief difference is that the second example has to call a named method to process the value of `i`, whereas the first example uses the nameless method (the code between curly brackets) to process `i`.

Look Familiar?

Now that you know what a block is, you may notice that you've seen them before. Many times. For example, you previously used `do..end` blocks to iterate over ranges like this:

```
(1..3).each do |i|
    puts(i)
end
```

You have also used `do..end` blocks to iterate over arrays (see *for_each2.rb* on page 69):

```
arr = ['one','two','three','four']
arr.each do |s|
    puts(s)
end
```

And you have executed a block repeatedly by passing it to the `loop` method (see *3loops.rb* on page 75):

```
i=0
loop {
    puts(arr[i])
    i+=1
    if (i == arr.length) then
        break
    end
}
```

The previous `loop` example is notable for two things: It has no list of items (such as an array or a range of values) to iterate over, and it is pretty darn ugly. These two features are not entirely unrelated! The `loop` method is part of the Kernel class, which is "automatically" available to your programs. Because it has no "end value," it will execute the block forever unless you explicitly break out of it using the `break` keyword. Usually there are more elegant ways to perform this kind of iteration—by iterating over a sequence of values with a finite range.

Blocks and Arrays

Blocks are commonly used to iterate over arrays. The Array class, consequently, provides a number of methods to which blocks are passed.

One useful method is called `collect`; this passes each element of the array to a block and creates a new array to contain each of the values returned by the block. Here, for example, a block is passed to each of the integers in an array (each integer is assigned to the variable `x`); the block doubles its value and returns it. The `collect` method creates a new array containing each of the returned integers in sequence:

2blocks.rb

```
b3 = [1,2,3].collect{|x| x*2}
```

The previous example assigns this array to `b3`:

```
[2,4,6]
```

In the next example, the block returns a version of the original strings in which each initial letter is capitalized:

```
b4 = ["hello","good day","how do you do"].collect{|x| x.capitalize }
```

So, `b4` is now as follows:

```
["Hello", "Good day", "How do you do"]
```

The `each` method of the Array class may look rather similar to `collect`; it too passes each array element in turn to be processed by the block. However, unlike `collect`, the `each` method does not create a new array to contain the returned values:

```
b5 = ["hello","good day","how do you do"].each{|x| x.capitalize }
```

This time, `b5` is unchanged:

```
["hello", "good day", "how do you do"]
```

Recall, however, that some methods—notably those ending with an exclamation mark (!)—actually alter the original objects rather than yielding new values. If you wanted to use the each method to capitalize the strings in the original array, you could use the capitalize! method:

```
b6 = ["hello","good day", "how do you do"].each{|x| x.capitalize! }
```

So, b6 is now as follows:

```
["Hello", "Good day", "How do you do"]
```

With a bit of thought, you could also use a block to iterate over the characters in a string. First, you need to split off each character from a string. This can be done using the split method of the String class like this:

```
"hello world".split(//)
```

The split method divides a string into substrings based on a delimiter and returns an array of these substrings. Here // is a regular expression that defines a zero-length string; this has the effect of returning a single character, so you end up creating an array of all the characters in the string. You can now iterate over this array of characters, returning a capitalized version of each:

```
a = "hello world".split(//).each{ |x| newstr << x.capitalize }
```

At each iteration, a capitalized character is appended to newstr, and the following is displayed:

```
H
HE
HEL
HELL
HELLO
HELLO
HELLO W
HELLO WO
HELLO WOR
HELLO WORL
HELLO WORLD
```

Because you are using the capitalize method here (with no terminating ! character), the characters in the array a remain as they began, all lowercase, since the capitalize method does not alter the receiver object (here the receiver objects are the characters passed into the block).

Be aware, however, that this code would not work if you were to use the `capitalize!` method to modify the original characters. This is because `capitalize!` returns `nil` when no changes are made, so when the space character is encountered, `nil` would be returned, and your attempt to append (`<<`) a `nil` value to the string `newstr` would fail.

You could also capitalize a string using the `each_byte` method. This iterates through the string characters, passing each byte to the block. These bytes take the form of ASCII codes. So, "hello world" would be passed in the form of these numeric values: `104 101 108 108 111 32 119 111 114 108 100`.

Obviously, you can't capitalize an integer, so you need to convert each ASCII value to a character. The `chr` method of String does this:

```
a = "hello world".each_byte{|x| newstr << (x.chr).capitalize }
```

Procs and Lambdas

In the examples up to now, blocks have been used in cahoots with methods. This has been a requirement since nameless blocks cannot have an independent existence in Ruby. You cannot, for example, create a stand-alone block like this:

```
{|x| x = x*10; puts(x)}        # This is not allowed!
```

This is one of the exceptions to the rule that "everything in Ruby is an object." A block clearly is not an object. Every object is created from a class, and you can find an object's class by calling its `class` method.

Do this with a hash, for example, and the class name "Hash" will be displayed:

```
puts({1=>2}.class)
```

Try this with a block, however, and you will only get an error message:

```
puts({|i| puts(i)}.class) #<= error!
```

Block or Hash?

Ruby uses curly brackets to delimit both blocks and hashes. So, how can you (and Ruby) tell which is which? The answer, basically, is that it's a hash when it *looks* like a hash, and otherwise it's a block. A hash looks like a hash when curly brackets contain key-value pairs:

```
puts( {1=>2}.class )     #<= Hash
```

or when they are empty:

block_or_hash.rb

```
puts( {}.class )          #<= Hash
```

However, if you omit the parentheses, there is an ambiguity. Is this an empty hash, or is it a block associated with the `puts` method?

```
puts{}.class
```

Frankly, I have to admit I don't know the answer to that question, and I can't get Ruby to tell me. Ruby accepts this as valid syntax but does not, in fact, display anything when the code executes. So, how about this?

```
print{}.class
```

Once again, this prints nothing at all in Ruby 1.9, but in Ruby 1.8 it displays `nil` (not, you will notice, the actual *class* of `nil`, which is NilClass, but `nil` itself). If you find all this confusing (as I do!), just remember that this can all be clarified by the judicious use of parentheses:

```
print( {}.class )    #<= Hash
```

Creating Objects from Blocks

Although blocks may not be objects by default, they can be "turned into" objects. There are three ways of creating objects from blocks and assigning them to variables—here's how:

proc_create.rb

```
a = Proc.new{|x| x = x*10; puts(x) }    #=> Proc
b = lambda{|x| x = x*10; puts(x) }      #=> Proc
c = proc{|x| x.capitalize! }            #=> Proc
```

In each of the three cases, you will end up creating an instance of the Proc class—which is the Ruby "object wrapper" for a block.

Let's take a look at a simple example of creating and using a Proc object. First, you can create an object calling `Proc.new` and passing to it a block as an argument:

3blocks.rb

```
a = Proc.new{|x| x = x*10; puts(x)}
```

Second, you can execute the code in the block to which `a` refers using the Proc class's `call` method with one or more arguments (matching the block parameters) to be passed into the block; in the previous code, you could pass an integer such as 100, and this would be assigned to the block variable `x`:

```
a.call(100)        #=> 1000
```

Finally, you can also create a Proc object by calling the `lambda` or `proc` methods, which are supplied by the Kernel class. The name `lambda` is taken from the Scheme (Lisp) language and is a term used to describe an anonymous method, or *closure*.

```
b = lambda{|x| x = x*10; puts(x) }
b.call(100)            #=> 1000

c = proc{|x| x.capitalize! }
c1 = c.call( "hello" )
puts( c1 )             #=> Hello
```

Here is a slightly more complicated example that iterates over an array of strings, capitalizing each string in turn. The array of capitalized strings is then assigned to the `d1` variable:

```
d = lambda{|x| x.capitalize! }
d1 = ["hello","good day","how do you do"].each{ |s| d.call(s)}
puts(d1.inspect)        #=> ["Hello", "Good day", "How do you do"]
```

There is one important difference between creating a Proc object using `Proc.new` and creating a Proc object using a `lambda` method—`Proc.new` does not check that the number of arguments passed to the block matches the number of block parameters. `lambda` does. The behavior of the `proc` method is different in Ruby 1.8 and 1.9. In Ruby 1.8, `proc` is equivalent to `lambda`—it checks the number of arguments. In Ruby 1.9, `proc` is equivalent to `Proc.new`—it does *not* check the number of arguments:

proc_lamba.rb

```
a = Proc.new{|x,y,z| x = y*z; puts(x) }
a.call(2,5,10,100)        # This is not an error

b = lambda{|x,y,z| x = y*z; puts(x) }
b.call(2,5,10,100)        # This is an error

puts('---Block #2---' )
c = proc{|x,y,z| x = y*z; puts(x) }
c.call(2,5,10,100)        # This is an error in Ruby 1.8
                          # Not an error in Ruby 1.9
```

What Is a Closure?

A *closure* is a function that has the ability to store (that is, to "enclose") values of local variables within the scope in which the block was created (think of this as the block's "native scope"). Ruby's blocks are closures. To understand this, look at this example:

block_closure.rb

```
x = "hello world"

ablock = Proc.new { puts( x ) }
```

```
def aMethod( aBlockArg )
    x = "goodbye"
    aBlockArg.call
end

puts( x )
ablock.call
aMethod( ablock )
ablock.call
puts( x )
```

Here, the value of the local variable `x` is "hello world" within the scope of `ablock`. Inside `aMethod`, however, a local variable named `x` has the value "goodbye." In spite of that, when `ablock` is passed to `aMethod` and called within the scope of `aMethod`, it prints "hello world" (that is, the value of `x` within the block's native scope) rather than "goodbye," which is the value of `x` within the scope of `aMethod`. The previous code, therefore, only ever prints "hello world."

NOTE *See "Digging Deeper" on page 175 for more on closures.*

yield

Let's see a few more blocks in use. The *4blocks.rb* program introduces something new, namely, a way of executing a nameless block when it is passed to a method. This is done using the keyword `yield`. In the first example, I define this simple method:

4blocks.rb

```
def aMethod
    yield
end
```

It doesn't really have any code of its own. Instead, it expects to receive a block, and the `yield` keyword causes the block to execute. This is how I pass a block to it:

```
aMethod{ puts( "Good morning" ) }
```

Notice that this time the block is not passed as a named argument. It would be an error to try to pass the block between parentheses, like this:

```
aMethod(  { puts( "Good morning" ) }  )     # This won't work!
```

Instead, you simply put the block right next to the method to which you are passing it, just as you did in the first example in this chapter. That method receives the block without having to declare a named parameter for it, and it calls the block with `yield`.

Here is a slightly more useful example:

```
def caps( anarg )
    yield( anarg )
end

caps( "a lowercase string" ){ |x| x.capitalize! ; puts( x ) }
```

Here the `caps` method receives one argument, `anarg`, and passes this argument to a nameless block, which is then executed by `yield`. When I call the `caps` method, I pass it a string argument (`"a lowercase string"`) using the normal parameter-passing syntax. The nameless block is passed *after the end* of the parameter list.

When the `caps` method calls `yield( anarg )`, then the string argument is passed into the block; it is assigned to the block variable `x`. This capitalizes it and displays it with `puts( s )`, which shows that the initial letter has been capitalized: "A lowercase string."

Blocks Within Blocks

You've already seen how to use a block to iterate over an array. In the next example (also in *4blocks.rb*), I use one block to iterate over an array of strings, assigning each string in turn to the block variable `s`. A second block is then passed to the `caps` method in order to capitalize the string:

```
["hello","good day","how do you do"].each{
    |s|
    caps( s ){ |x| x.capitalize!
        puts( x )
    }
}
```

This results in the following output:

```
Hello
Good day
How do you do
```

Passing Named Proc Arguments

Up to now, you have passed blocks to procedures either anonymously (in which case the block is executed with the `yield` keyword) or in the form of a named argument, in which case it is executed using the `call` method. There is another way to pass a block. When the last argument in a method's list of parameters is preceded by an ampersand (`&`), it is considered to be a Proc object. This gives you the option of passing an anonymous block to a

procedure using the same syntax as when passing a block to an iterator, and yet the procedure itself can receive the block as a named argument. Load *5blocks.rb* to see some examples of this.

First, here is a reminder of the two ways you've already seen of passing blocks. This method has three parameters, `a`, `b`, and `c`:

5blocks.rb

```
def abc( a, b, c )
    a.call
    b.call
    c.call
    yield
end
```

You call this method with three named arguments (which here happen to be blocks but could, in principle, be anything) plus an unnamed block:

```
a = lambda{ puts "one" }
b = lambda{ puts "two" }
c = proc{ puts "three" }
abc(a, b, c ){ puts "four" }
```

The `abc` method executes the named block arguments using the `call` method and the unnamed block using the `yield` keyword. The results are shown in the `#=>` comments here:

```
a.call    #=> one
b.call    #=> two
c.call    #=> three
yield     #=> four
```

The next method, `abc2`, takes a single argument, `&d`. The ampersand here is significant because it indicates that the `&d` parameter is a block. Instead of using the `yield` keyword, the `abc2` method is able to execute the block using the name of the argument (without the ampersand):

```
def abc2( &d )
    d.call
end
```

So, a block argument with an ampersand is called in the same way as one without an ampersand. However, there is a difference in the way the object matching that argument is passed to the method. To match an ampersand-argument, an unnamed block is passed by appending it to the method name:

```
abc2{ puts "four" }
```

You can think of ampersand-arguments as type-checked block parameters. Unlike normal arguments (without an ampersand), the argument cannot match any type; it can match only a block. You cannot pass some other sort of object to `abc2`:

```
abc2( 10 )    # This won't work!
```

The `abc3` method is essentially the same as the `abc` method except it specifies a fourth formal block-typed argument (`&d`):

```
def abc3( a, b, c, &d)
```

The arguments `a`, `b`, and `c` are called, while the argument `&d` may be either called or yielded, as you prefer:

```
def abc3( a, b, c, &d)
    a.call
    b.call
    c.call
    d.call        # first call block &d
    yield         # then yield block &d
end
```

This means the calling code must pass to this method three formal arguments plus a block, which may be nameless:

```
abc3(a, b, c){ puts "five" }
```

The previous method call would result in this output (bearing in mind that the final block argument is executed twice since it is both called and yielded):

```
one
two
three
five
five
```

You can also use a preceding ampersand in order to pass a named block to a method when the receiving method has no matching named argument, like this:

```
myproc = proc{ puts("my proc") }
abc3(a, b, c, &myproc )
```

An ampersand block variable such as `&myproc` in the previous code may be passed to a method even if that method does not declare a matching variable in its argument list. This gives you the choice of passing either an unnamed block or a Proc object:

```
xyz{ |a,b,c| puts(a+b+c) }
xyz( &myproc )
```

Be careful, however! Notice in one of the previous examples, I have used block parameters (`|a,b,c|`) with the same names as the three local variables to which I previously assigned Proc objects: `a`, `b`, `c`:

```
a = lambda{ puts "one" }
b = lambda{ puts "two" }
c = proc{ puts "three" }
xyz{ |a,b,c| puts(a+b+c) }
```

In principle, block parameters should be visible only within the block itself. However, it turns out that assignment to block parameters has profoundly different effects in Ruby 1.8 and Ruby 1.9. Let's look first at Ruby 1.8. Here, assignment to block parameters can initialize the values of any local variables with the same name within the block's native scope (see "What Is a Closure?" on page 163).

Even though the variables in the `xyz` method are named `x`, `y`, and `z`, it turns out that the integer assignments in that method are actually made to the variables `a`, `b`, and `c` when this block:

```
{ |a,b,c| puts(a+b+c) }
```

is passed the values of `x`, `y`, and `z`:

```
def xyz
    x = 1
    y = 2
    z = 3
    yield( x, y, z )   # 1,2,3 assigned to block parameters a,b,c
end
```

As a consequence, the Proc variables `a`, `b`, and `c` within the block's native scope (the main scope of my program) are initialized with the integer values of the block variables `x`, `y`, and `z` once the code in the block has been run. So, `a`, `b`, and `c`, which began as Proc objects, end up as integers.

In Ruby 1.9, on the contrary, the variables inside the block are sealed off from the variables declared outside the block. So, the values of the `xyz` method's `x`, `y`, and `z` variables are not assigned to the block's `a`, `b`, and `c` parameters. That means once the block has executed, the values of the `a`, `b`, and `c` variables declared outside that method are unaffected: They began as Proc objects, and they end up as Proc objects.

Now let's suppose you execute the following code, remembering that `a`, `b`, and `c` are Proc objects at the outset:

```
xyz{ |a,b,c| puts(a+b+c) }
puts( a, b, c )
```

In Ruby 1.8, the `puts` statement shown earlier displays the end values of `a`, `b`, and `c`, showing that they have been initialized with the integer values that were passed into the block when it was yielded (`yield( x, y, z )`) in the `xyz` method. As a consequence, they are now integers:

```
1
2
3
```

But in Ruby 1.9, `a`, `b`, and `c` are not initialized by the block parameters and remain, as they began, as Proc objects:

```
#<Proc:0x2b65828@C:/bookofruby/ch10/5blocks.rb:36 (lambda)>
#<Proc:0x2b65810@C:/bookofruby/ch10/5blocks.rb:37 (lambda)>
#<Proc:0x2b657f8@C:/bookofruby/ch10/5blocks.rb:38>
```

This behavior can be difficult to understand, but it is worth taking the time to do so. The use of blocks is commonplace in Ruby, and it is important to know how the execution of a block may (or may not) affect the values of variables declared outside the block. To clarify this, try the simple program in *6blocks.rb*:

6blocks.rb

```
a = "hello world"

def foo
    yield 100
end

puts( a )
foo{ |a| puts( a ) }

puts( a )
```

Here `a` is a string within the scope of the main program. A different variable with the same name, `a`, is declared in the block, which is passed to `foo` and yielded. When it is yielded, an integer value, 100, is passed into the block, causing the block's parameter, `a`, to be initialized to 100. The question is, does the initialization of the block argument, `a`, also initialize the string variable, `a`, in the main scope? And the answer is, *yes* in Ruby 1.8 but *no* in Ruby 1.9.

Ruby 1.8 displays this:

```
hello world
100
100
```

Ruby 1.9 displays this:

```
hello world
100
hello world
```

If you want to make sure that block parameters do not alter the values of variables declared outside the block, no matter which version of Ruby you use, just ensure that the block parameter names do not duplicate names used elsewhere. In the current program, you can do this simply by changing the name of the block argument to ensure that it is unique to the block:

```
foo{ |b| puts( b ) }    # the name 'b' is not used elsewhere
```

This time, when the program is run, Ruby 1.8 and Ruby 1.9 both produce the same results:

```
hello world
100
hello world
```

This is an example of one of the pitfalls into which it is all too easy to fall in Ruby. As a general rule, when variables share the same scope (for example, a block declared within the scope of the main program here), it is best to make their names unique in order to avoid any unforeseen side effects. For more on scoping, see “Blocks and Local Variables” on page 177.

Precedence Rules

Blocks within curly brackets have stronger precedence than blocks within `do` and `end`. Let’s see what that means in practice. Consider these two examples:

```
foo bar do |s| puts( s ) end
foo bar{ |s| puts(s) }
```

Here, `foo` and `bar` are both methods, and the code between curly brackets and `do` and `end` are blocks. So, to which of the two methods is each of these blocks passed? It turns out that the `do..end` block would be passed to the leftmost method, `foo`, whereas the block in curly brackets would be sent to the rightmost method, `bar`. This is because curly brackets are said to have higher precedence than `do` and `end`.

Consider this program:

precedence.rb

```
def foo( b )
    puts("---in foo---")
    a = 'foo'
    if block_given?
        puts( "(Block passed to foo)" )
        yield( a )
    else
        puts( "(no block passed to foo)" )
    end
    puts( "in foo, arg b = #{b}" )
    return "returned by " << a
end

def bar
    puts("---in bar---")
    a = 'bar'
    if block_given?
        puts( "(Block passed to bar)" )
        yield( a )
    else
        puts( "(no block passed to bar)" )
    end
    return "returned by " << a
end

foo bar do |s| puts( s ) end       # 1) do..end block
foo bar{ |s| puts(s) }             # 2) {..} block
```

Here the `do..end` block has lower precedence, and the method `foo` is given priority. This means both `bar` and the `do..end` block are passed to `foo`. Thus, these two expressions are equivalent:

```
foo bar do |s| puts( s ) end
foo( bar ) do |s| puts( s ) end
```

A curly bracket block, on the other hand, has stronger precedence, so it tries to execute immediately and is passed to the first possible receiver method (`bar`). The result (that is, the value returned by `bar`) is then passed as an argument to `foo`, but this time, `foo` does not receive the block itself. Thus, the two following expressions are equivalent:

```
foo bar{ |s| puts(s) }
foo( bar{ |s| puts(s) } )
```

If you are confused by all this, take comfort in that you are not alone! The potential ambiguities result from the fact that, in Ruby, the parentheses around argument lists are optional. As you can see from the alternative versions I gave earlier, the ambiguities disappear when you use parentheses.

NOTE *A method can test whether it has received a block using the* `block_given?` *method. You can find examples of this in the* precedence.rb *program.*

Blocks as Iterators

As mentioned earlier, one of the primary uses of blocks in Ruby is to provide iterators to which a range or list of items can be passed. Many standard classes such as Integer and Array have methods that can supply items over which a block can iterate. For example:

```
3.times{ |i| puts( i ) }
[1,2,3].each{|i| puts(i) }
```

You can, of course, create your own iterator methods to provide a series of values to a block. In the *iterate1.rb* program, I have defined a simple `timesRepeat` method that executes a block a specified number of times. This is similar to the `times` method of the Integer class except it begins at index 1 rather than at index 0 (here the variable `i` is displayed in order to demonstrate this):

iterate1.rb

```
def timesRepeat( aNum )
    for i in 1..aNum do
        yield i
    end
end
```

Here is an example of how this method might be called:

```
timesRepeat( 3 ){  |i| puts("[#{i}] hello world") }
```

This displays the following:

```
[1] hello world
[2] hello world
[3] hello world
```

I've also created a `timesRepeat2` method to iterate over an array:

```
def timesRepeat2( aNum, anArray )
    anArray.each{ |anitem|
        yield( anitem )
    }
end
```

This could be called in this manner:

```
timesRepeat2( 3, ["hello","good day","how do you do"] ){ |x| puts(x) }
```

This displays the following:

```
hello
good day
how do you do
```

Of course, it would be better (truer to the spirit of object orientation) if an object itself contained its own iterator method. I've implemented this in the next example. Here I have created MyArray, a subclass of Array:

```
class MyArray < Array
```

It is initialized with an array when a new MyArray object is created:

```
def initialize( anArray )
    super( anArray )
end
```

It relies upon its own `each` method (an object refers to itself as `self`), which is provided by its ancestor, Array, to iterate over the items in the array, and it uses the `times` method of Integer to do this a certain number of times. This is the complete class definition:

iterate2.rb

```
class MyArray < Array
    def initialize( anArray )
        super( anArray )
    end

    def timesRepeat( aNum )
        aNum.times{           # start block 1...
             | num |
             self.each{       # start block 2...
                  | anitem |
                  yield( "[#{num}] :: '#{anitem}'" )
             }                # ...end block 2
        }                     # ...end block 1
    end
end
```

Notice that, because I have used two iterators (`aNum.times` and `self.each`), the `timesRepeat` method comprises two nested blocks. This is an example of how you might use this:

```
numarr = MyArray.new( [1,2,3] )
numarr.timesRepeat( 2  ){ |x| puts(x) }
```

This would output the following:

```
[0] :: '1'
[0] :: '2'
[0] :: '3'
[1] :: '1'
[1] :: '2'
[1] :: '3'
```

In *iterate3.rb*, I have set myself the problem of defining an iterator for an array containing an arbitrary number of subarrays, in which each subarray has the same number of items. In other words, it will be like a table or matrix with a fixed number of rows and a fixed number of columns. Here, for example, is a multidimensional array with three "rows" (subarrays) and four "columns" (items):

iterate3.rb

```
multiarr =
[ ['one','two','three','four'],
  [1,    2,    3,      4     ],
  [:a,   :b,   :c,    :d     ]
]
```

I've tried three alternative versions of this. The first version suffers from the limitation of only working with a predefined number (here 2 at indexes [0] and [1]) of "rows" so it won't display the symbols in the third row:

```
multiarr[0].length.times{|i|
    puts(multiarr[0][i], multiarr[1][i])
}
```

The second version gets around this limitation by iterating over each element (or "row") of `multiarr` and then iterating along each item in that row by obtaining the row length and using the Integer's `times` method with that value. As a result, it displays the data from all three rows:

```
multiarr.each{ |arr|
    multiarr[0].length.times{|i|
        puts(arr[i])
    }
}
```

The third version reverses these operations: The outer block iterates along the length of row 0, and the inner block obtains the item at index `i` in each row. Once again, this displays the data from all three rows:

```
multiarr[0].length.times{|i|
    multiarr.each{ |arr|
        puts(arr[i])
    }
}
```

However, although versions 2 and 3 work in a similar way, you will find that they iterate through the items in a different order. Version 2 iterates through each complete row one at a time. Version 3 iterates down the items in each column. Run the program to verify that. You could try creating your own subclass of Array and adding iterator methods like this—one method to iterate through the rows in sequence and one to iterate through the columns.

DIGGING DEEPER

Here we look at important differences between block scoping in Ruby 1.8 and 1.9 and also learn about returning blocks from methods.

Returning Blocks from Methods

Earlier, I explained that blocks in Ruby may act as closures. A closure may be said to enclose the "environment" in which it is declared. Or, to put it another way, it carries the values of local variables from its original scope into a different scope. The example I gave previously showed how the block named `ablock` captures the value of the local variable `x`:

block_closure.rb

```
x = "hello world"
ablock = Proc.new { puts( x ) }
```

It is then able to "carry" that variable into a different scope. Here, for example, `ablock` is passed to `aMethod`. When `ablock` is called inside that method, it runs the code `puts( x )`. This displays "hello world" and not "goodbye":

```
def aMethod( aBlockArg )
    x = "goodbye"
    aBlockArg.call            #=> "hello world"
end
```

In this particular example, this behavior may seem like a curiosity of no great interest. In fact, block/closures can be used more creatively.

For example, instead of creating a block and sending it to a method, you could create a block *inside a method* and return that block to the calling code. If the method in which the block is created happens to take an argument, the block could be initialized with that argument.

This gives you a simple way of creating multiple blocks from the same "block template," each instance of which is initialized with different data. Here, for example, I have created two blocks and assigned them to the variables `salesTax` and `vat`, each of which calculates results based on different values (0.10) and (0.175):

block_closure2.rb

```
def calcTax( taxRate )
    return lambda{
        |subtotal|
            subtotal * taxRate
    }
end

salesTax = calcTax( 0.10 )
vat = calcTax( 0.175 )
```

```
print( "Tax due on book = ")
print( salesTax.call( 10 ) )       #=> 1.0

print( "\nVat due on DVD = ")
print( vat.call( 10 ) )            #=> 1.75
```

Blocks and Instance Variables

One of the less obvious features of blocks is the way in which they use variables. If a block may truly be regarded as a nameless function or method, then, logically, it should be able to contain its own local variables and have access to the instance variables of the object to which the block belongs.

Let's look first at instance variables. Load the *closures1.rb* program. This provides another illustration of a block acting as a closure—by capturing the values of the local variables in the scope in which it was created. Here I have created a block using the `lambda` method:

closures1.rb

```
aClos = lambda{
    @hello << " yikes!"
}
```

This block appends the string " yikes!" to the instance variable `@hello`. Notice that at this stage in the proceedings, no value has previously been assigned to `@hello`. I have, however, created a separate method, `aFunc`, which does assign a value to a variable called `@hello`:

```
def aFunc( aClosure )
    @hello = "hello world"
    aClosure.call
end
```

When I pass my block (the `aClosure` argument) to the `aFunc` method, the method brings `@hello` into being. I can now execute the code inside the block using the `call` method. And sure enough, the `@hello` variable contains the "hello world" string. The same variable can also be used by calling the block outside of the method. Indeed, now, by repeatedly calling the block, I will end up repeatedly appending the string " yikes!" to `@hello`:

```
aFunc(aClos)      #<= @hello = "hello world yikes!"
aClos.call        #<= @hello = "hello world yikes! yikes!"
aClos.call        #<= @hello = "hello world yikes! yikes! yikes!"
aClos.call        # ...and so on
```

If you think about it, this is not too surprising. After all, `@hello` is an instance variable, so it exists within the scope of an object. When you run a Ruby program, an object called `main` is automatically created. So, you should expect any instance variable created within that object (the program) to be available to everything inside it.

The question now arises: What would happen if you were to send the block to a method of some *other* object? If that object has its own instance variable, `@hello`, which variable will the block use—the `@hello` from the scope in which the block was created or the `@hello` from the scope of the object in which the block is called? Let's try that. You'll use the same block as before, except this time it will display a bit of information about the object to which the block belongs and the value of `@hello`:

```
aClos = lambda{
    @hello << " yikes!"
    puts("in #{self} object of class #{self.class}, @hello = #{@hello}")
}
```

Now, create a new object from a new class (X), and give it a method that will receive the block `b` and call the block:

```
class X
    def y( b )
        @hello = "I say, I say, I say!!!"
        puts( "   [In X.y]" )
        puts("in #{self} object of class #{self.class}, @hello = #{@hello}")
        puts( "   [In X.y] when block is called..." )
        b.call
    end
end

x = X.new
```

To test it, just pass the block `aClos` to the `y` method of `x`:

```
x.y( aClos )
```

And this is what is displayed:

```
   [In X.y]
in #<X:0x32a6e64> object of class X, @hello = I say, I say, I say!!!
    [In X.y] when block is called...
in main object of class Object, @hello = hello world yikes! yikes! yikes!
yikes! yikes! yikes!
```

So, it is clear that the block executes in the scope of the object in which it was *created* (`main`) and retains the instance variable from that object even though the object in whose scope the block is *called* has an instance variable with the same name and a different value.

Blocks and Local Variables

Now let's see how a block/closure deals with local variables. In the *closures2.rb* program, I declare a variable, `x`, which is local to the context of the program:

closures2.rb

```
x = 3000
```

The first block/closure is called `c1`. Each time I call this block, it picks up the value of `x` defined outside the block (3,000) and returns `x + 100`:

```
c1 = proc{
    x + 100
}
```

Incidentally, even though this returns a value (in ordinary Ruby methods, the default value is the result of the last expression to be evaluated), in Ruby 1.9 you cannot explicitly use the `return` statement here like this:

```
return x + 1
```

If you do this, Ruby 1.9 throws a LocalJumpError exception. Ruby 1.8, on the other hand, does not throw an exception.

This block has no block parameters (that is, there are no "block-local" variables between upright bars), so when it is called with a variable, `someval`, that variable is discarded, unused. In other words, `c1.call(someval)` has the same effect as `c1.call()`.

So, when you call the block `c1`, it returns `x+100` (that is, 3,100); this value is then assigned to `someval`. When you call `c1` a second time, the same thing happens all over again, so once again `someval` is assigned 3,100:

```
someval=1000
someval=c1.call(someval); puts(someval)    #<= someval is now 3100
someval=c1.call(someval); puts(someval)    #<= someval is now 3100
```

NOTE *Instead of repeating the call to `c1`, as shown earlier, you could place the call inside a block and pass this to the `times` method of Integer like this:*

```
2.times{ someval=c1.call(someval); puts(someval) }
```

However, because it can be hard enough to work out what just one block is up to (such as the `c1` block here), I've deliberately avoided using any more blocks than are absolutely necessary in this program!

The second block is named `c2`. This declares the "block parameter" `z`. This too returns a value:

```
c2 = proc{
    |z|
    z + 100
}
```

However, this time the returned value can be reused since the block parameter acts like an incoming argument to a method—so when the value of `someval` is changed after it is assigned the return value of `c2`, this changed value is subsequently passed as an argument:

```
someval=1000
someval=c2.call(someval); puts(someval)      #<= someval is now 1100
someval=c2.call(someval); puts(someval)      #<= someval is now 1200
```

The third block, `c3`, looks, at first sight, pretty much the same as the second block, `c2`. In fact, the only difference is that its block parameter is called `x` instead of `z`:

```
c3 = proc{
    |x|
    x + 100
}
```

The name of the block parameter has no effect on the return value. As before, `someval` is first assigned the value 1,100 (that is, its original value, 1,000, plus the 100 added inside the block). Then, when the block is called a second time, `someval` is assigned the value 1,200 (its previous value, 1,100, plus 100 assigned inside the block).

But now look at what happens to the value of the local variable `x`. This was assigned 3,000 at the top of the unit. Remember that, in Ruby 1.8, an assignment to a block parameter can change the value of a variable with the same name in its surrounding context. In Ruby 1.8, then the local variable `x` changes when the block parameter `x` is changed. It now has the value, 1,100—that is, the value that the block parameter, `x`, last had when the `c3` block was called:

```
x = 3000
someval=1000
someval=c3.call(someval); puts(someval)    #=> 1100
someval=c3.call(someval); puts(someval)    #=> 1200
puts( x ) # Ruby 1.8, x = 1100. Ruby 1.9, x = 3000
```

Incidentally, even though block-local variables and block parameters can affect similarly named local variables outside the block in Ruby 1.8, the block variables themselves have no "existence" outside the block. You can verify this using the `defined?` keyword to attempt to display the type of variable if it is, indeed, defined:

```
print("x=[#{defined?(x)}],z=[#{defined?(z)}]")
```

This demonstrates that only `x`, and not the block variable `z`, is defined in the main scope:

```
x=[local-variable], z=[]
```

Matz, the creator of Ruby, has described the scoping of local variables within a block as "regrettable." Although Ruby 1.9 has addressed some issues, it is worth noting that one other curious feature of block scoping remains: Namely, local variables within a block are invisible to the method containing that block. This may be changed in future versions. For an example of this, look at this code:

local_var_scope.rb

```
def foo
    a = 100
    [1,2,3].each do |b|
        c = b
        a = b
        print("a=#{a}, b=#{b}, c=#{c}\n")
    end
    print("Outside block: a=#{a}\n")    # Can't print #{b} and #{c} here!!!
end
```

Here, the block parameter, `b`, and the block-local variable, `c`, are both visible only when inside the block. The block has access to both these variables and to the variable `a` (local to the `foo` method). However, outside of the block, `b` and `c` are inaccessible, and only `a` is visible.

Just to add to the confusion, whereas the block-local variable, `c`, and the block parameter, `b`, are both inaccessible outside the block in the previous example, they are accessible when you iterate a block with `for`, as in the following example:

```
def foo2
    a = 100
    for b in [1,2,3] do
        c = b
        a = b
        print("a=#{a}, b=#{b}, c=#{c}\n")
    end
    print("Outside block: a=#{a}, b=#{b}, c=#{b}\n")
end
```