Ruby Blocks & Closures

Sang Shin
Michèle Garoche
www.javapassion.com
"Learning is fun!"

Topics

- Blocks
  - What is a block?
  - How does a block look like?
  - How does a block get passed and executed?
  - Proc object
  - & (Ampersand) operator
  - lambda
  - Where do blocks get used?
- Closure
  - What is a closure?

What is a Block (or Code Block)?

What is a block?

- Block is basically a nameless function
  - You can think of it as a nameless chunk of code.
- You can pass a nameless function to another function (I will call it a target function in this presentation), and then that target function can invoke the passed-in nameless function
  - For example, a target function could perform iteration by passing one item at a time to the nameless function.

If you don't understand what is being said here, don't worry about it. We will explore this with examples!

How Does a Block Look Like?

How to Represent a Block?

- A block can be represented in two different formats - these two formats are functionally equivalent
- Convention is use { } for a single line block to use do ... end for multi-line block

```
puts "----First format of code block containing code
   fragment between { and }"
[1, 2, 3].each { puts "Life is good!" }

puts "----Second format of code block containing code
   fragment between do and end"
[1, 2, 3].each do
  puts "Life is good!"
end
```

How Does a Block Get Passed & Executed?

How a block is passed & executed

- When a method is invoked, a block can be passed (sometimes called attached)
- The *yield()* method in the invoked method (target function) executes the passed block

```
puts "----Define MyClass which invokes yield"
class MyClass
  def command()
    # yield will execute the attached block
    yield()
  end
end

puts "----Create object instance of MyClass"
m = MyClass.new
puts "----Call command method of the MyClass passing a block"
m.command {puts "Hello World!"}
```

How a block receive arguments

- A block can receive arguments - they are represented as comma-separated list at the beginning of the block, enclosed in pipe characters:

```
puts "----Define MyClass which invokes yield"
class MyClass
  def command1()
    # yield will execute the supplied block
    yield(Time.now)
  end
end

puts "----Create an object instance of MyClass"
m = MyClass.new

puts "----Call command1 method of the MyClass"
m.command1() {|x| puts "Current time is #{x}"}
```

Example: Block Receive Argument

- *each* method of *Array* class pass an element as an argument

```
puts "----First format of code block containing code
   fragment between { and }"
[1, 2, 3].each { |n| puts "Number #{n}" }

puts "----Second format of code block containing code
   fragment between do and end"
[1, 2, 3].each do |n|
  puts "Number #{n}"
end
```

How a block can receive arguments

- A block can receive multiple arguments

```
puts "---Define a method called testyield"
def testyield
  yield(1000, "Sang Shin")
  yield("Current time is", Time.now)
end

puts "----Call testyield method"
testyield { |arg1, arg2| puts "#{arg1} #{arg2}" }
```

- Result

```
---Define a method called testyield
----Call testyield method
1000 Sang Shin
Current time is Mon Jun 30 09:14:56 -0400 2008
```

Proc Objects

What is a Proc object?

- Proc objects (Procs) are blocks of code that have been converted to “callable” objects.
- Proc objects are considered as first-class objects in Ruby language because they can be
  - Created during runtime
  - Stored in data structures (saved in a variable)
  - Passed as arguments to other functions
  - Returned as the value of other functions.

How To Create and Execute a Proc Object?

- Use new keyword of Proc class passing a block to create a Proc object and use call method of the Proc object to execute it

```
puts "----Create a Proc object and call it"
say_hi = Proc.new { puts "Hello Sydney" }
say_hi.call

puts "----Create another Proc object and call it"
Proc.new { puts "Hello Boston"}.call
```

- Result

```
----Create a Proc object and call it
Hello Sydney
----Create another Proc object and call it
Hello Boston
```

How to pass a Proc object as an argument?

- Pass it just like any other Ruby object (like a String object) - hence the reason why Proc object is a first-class object in Ruby

```
puts "----Create a Proc object"
my_proc = Proc.new {|x| puts x}

puts "----Define a method that receives Proc object as an argument"
def foo (proc_param, b)
  proc_param.call(b)
end

puts "----Call a method that passes a Proc object as an arugment"
foo(my_proc, 'Sang Shin')
```

How to pass Arguments to the block (represented by the Proc object)?

- Pass arguments in a call method of the proc object

```
puts "----Create a Proc object"
my_proc = Proc.new {|x| puts x}

puts "----Define a method that receives Proc object as an argument"
def foo (proc_param, b)
  proc_param.call(b)
end

puts "----Call a method that passes a Proc object as an arugment"
foo(my_proc, 'Sang Shin')
```

How to Use a Proc object as a Return Value?

- Just like any other Ruby object

```
puts "----Define a method that returns Proc object as a return value"
def gen_times(factor)
    Proc.new {|n| n*factor }
end

puts "----Assign Proc object to local variables"
times3 = gen_times(3)

puts "----Execute the code block passing a parameter"
puts times3.call(3)
```

Proc Object works as a Closure

Proc Object works as a Closure

- Proc objects (Procs) are blocks of code that have been bound to a set of local variables. Once bound, the code may be called in different contexts and still access those variables.

```
def gen_times(factor)
  return Proc.new {|n| n*factor }   # factor is local variable in a block
end

times3 = gen_times(3)              # Proc object has factor variable set to 3
times5 = gen_times(5)              # Proc object has factor variable set to 5

times3.call(12)                    #=> 36 because 12 * 3 (factor) = 36
times5.call(5)                     #=> 25 because 5 * 5 (factor) = 25
times3.call(times5.call(4))        #=> 60 because (4 * 5  )  * 3  = 60
```

lambda

lambda and Proc

- *lambda* is equivalent to *Proc.new* - the following three statements are considered equivalent

```
say_hi = Proc.new { puts "Hello Sydney" }
say_hi = lambda { puts "Hello Sydney" }
say_hi = proc { puts "Hello Sydney" }
```

- There are a couple of differences, however

Difference #1 between *lambda* and *Proc*

- Argument checking: A *Proc* object created from using *lambda* checks the number of arguments passed and if they do not match, throws *ArgumentError* exception

```
puts "----Create Proc object using lambda"
lamb = lambda {|x, y| puts x + y}

puts "----Create Proc object using Proc.new"
pnew = Proc.new {|x, y| puts x + y}

puts "----Send 3 arguments, should work"
pnew.call(2, 4, 11)

puts "----Send 3 arguments, throws an ArgumentError"
lamb.call(2, 4, 11)
```

Difference #2 between lambda and Proc

- How return is handled from the Proc
  - A return from *Proc.new* returns from the enclosing method (acting just like a return from a block).
  - A return from lambda acts more conventionally, returning to its caller.

```
def foo
  f = Proc.new { return "return from foo from inside proc" }
  f.call # control leaves foo here
  return "return from foo"
end
def bar
  f = lambda { return "return from lambda" }
  f.call # control does not leave bar here
  return "return from bar"
end
puts foo # prints "return from foo from inside proc"
puts bar # prints "return from bar"
```

& (Ampersand) Operator

How & (Ampersand) is used?

- The ampersand operator (&) can be used to explicitly convert between blocks and Procs
- Conversion from a block to a Proc
  - If an ampersand (&) is prepended to the last argument in the argument list of a method, the block attached to this method is converted to a Proc object and gets assigned to that last argument.
- Conversion from a Proc to a block
  - Another use of the ampersand is the other-way conversion - converting a Proc into a block. This is very useful because many of Ruby's great built-ins, and especially the iterators, expect to receive a block as an argument, and sometimes it's much more convenient to pass them a Proc.

Conversion from a Block to a Proc

- The method receives a block as a Proc object

```
puts "----The attached block is passed as the last argument
    in the form of Proc object"
def my_method_ampersand(a, &f)
  # the block can be accessed through f
  f.call(a)

  # but yield also works !
  yield(a)
end

puts "----Call a method with a block"
my_method_ampersand("Korea") {|x| puts x}
```

Conversion from a Proc to a Block

- Pass a Proc with & preceded

```
puts "----Create a Proc object"
say_hi = Proc.new { |x| puts "#{x} Hello Korea" }

puts "----Define a method which expects a block NOT Proc object"
def do_it_with_block
  if block_given?
    yield(1)
  end
end

puts "----Call do_it_with_block method which expects a block,
    convert Proc object to a block"
do_it_with_block(&say_hi)
```

Where Do Blocks Get Used?

Blocks Usage Examples

- Iteration

  [1, 2, 3].each {|item| puts item}
- Resource management

  file_contents = open(file_name) { |f| f.read }
- Callbacks

```
widget.on_button_press do
    puts “Button is pressed”
end
```

What is a Ruby Closure?

What is a Ruby Closure?

- In Ruby, a Proc object behaves as a Closure
  - A Proc object maintains all the context in which the block was defined: the value of self, and the methods, variables, and constants in scope. This context is called scope information
  - A block of the Proc object can still use all original scope information such as the variables even if the environment in which it was defined would otherwise have disappeared.

Ruby Closure Example

```
# Define a method that returns a Proc object
def ntimes(a_thing)
  return proc { |n| a_thing * n }
end

# The a_thing is set to value 23 in a block.
p1 = ntimes(23)

# Note that ntimes() method has returned.  The block still
# has access to a_thing variable.

# Now execute the block.  Note that the a_thing is still set to
# 23 and the code in the block can access it, so the results is set 69 and 92
puts p1.call(3)      #    69
puts p1.call(4)      #    92
```

Thank you!

We do Instructor-led Codecamps!
http://www.javapassion.com/codecamps
"Learn with Passion!"