Ruby Blocks & Closures

Sang Shin
JPassion.com
"Code with Passion!"

Topics

- Blocks
  - What is a block?
  - How does a block look like?
  - How does a block get passed and executed?
- Proc
- & (Ampersand)
- Lambda
- Closure

What is a Block (Code Block)?

What is a block (code block)?

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

```
# each is a method of Array class
[1, 2, 3].each { |n| puts "Number #{n}" }
```

Target method

Block

How Does a Block Look Like?

How to Represent (Create) a Block?

- A block can be represented (created) in two different formats - these two formats are functionally equivalent
- Convention
  - Use { } for a single line block
  - 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 and Executed?

How a block is passed & executed

- When a method is invoked, a block can be “passed” (sometimes called “attached”) – think of it as a special argument to the method
- The *yield()* method in the invoked method (target method) executes the passed-in block

```
puts "----Define MyClass which invokes yield"
class MyClass
  def command()
    # yield will execute the passed-in block
    yield    # same as 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, as a nameless function, itself 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)  # pass an argument to a block
  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 passes each element as an argument to a block

```
[1, 2, 3].each { |n| puts "Number #{n}" }

[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")               # pass two arguments to a block
  yield("Current time is", Time.now)  # pass two arguments to a block
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
```

Lab:

Exercise 1: Ruby Blocks
5512_ruby_blocks.zip

Proc Objects

What is a Proc object?

- Proc objects (called as Proc's) are blocks of code that have been converted into objects
  - These objects are “callable” (“executable”)
- Proc objects are considered as first-class objects in Ruby language (just like String object) because they can be
  - Created during run-time
  - Assigned to a variable
  - Passed as an argument to functions
  - Returned as the return value of functions
- Besides, the block of code captured in the Proc object can be called
  - Via “call” method

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" }   # Create a Proc object
say_hi.call                                  # Call Proc object

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 from a block"
my_proc = Proc.new {|x| puts x}

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

puts "----Call a method that passes a Proc object as an argument"
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
- Note: This is the same code in the previous slide, but emphasizing a different point

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

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

puts "----Call a method that passes a Proc object as an argument"
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 }   # return a Proc object
end

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

puts "----Execute the code block passing an argument"
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)
   mynum = factor *2;
   Proc.new {|n| n*factor + mynum}
end

 times3 = gen_times(3)            # factor set to 3 and mynum set to 6
 times5 = gen_times(5)            # factor set to 5 and mynum set to 10

 times3.call(12)                  #=> 42 because 12(n) * 3(factor) + 6(mynum)
 times5.call(4)                   #=> 30 because 4(n) * 5(factor)  + 10(mynum)
 times3.call(times5.call(4))     #=> 96 because 30(n) * 3(factor) + 6(mynum)
```

lambda

lambda and proc

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

```
say_hi = Proc.new { |x| puts "Hello #{x}" }
say_hi = proc {  |x|puts "Hello #{x}" }

say_hi = lambda {  |x| puts "Hello #{x}" }
say_hi = ->(x) {puts "Hello #{x}" }   # New syntax from Ruby 1.9
```

Lab:

Exercise 2: Proc & Lambda 5512_ruby_blocks.zip

& (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 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}
```

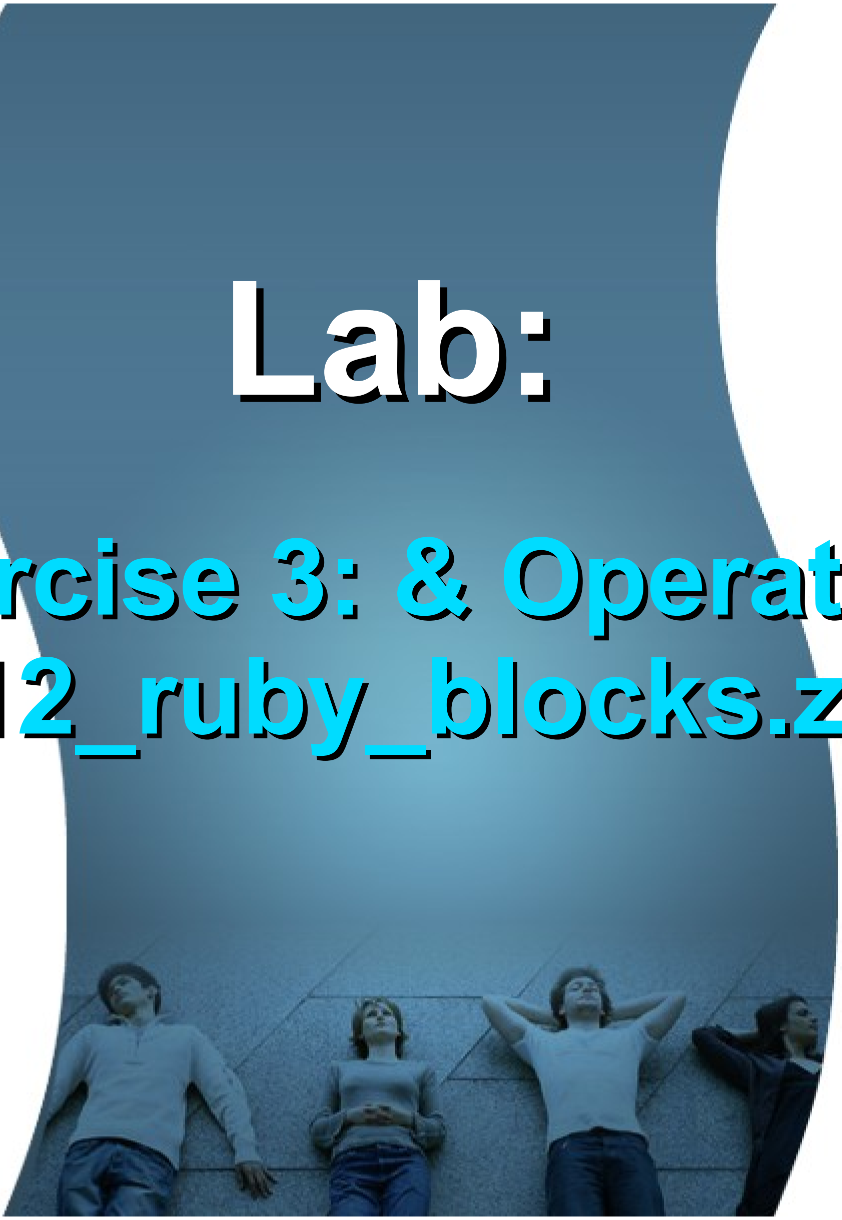
Conversion from a Proc to a Block

- 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)
```



Lab:

Exercise 3: & Operator
5512_ruby_blocks.zip

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
```

Lab:

Exercise 4: Ruby Blocks & Iterators
5512_ruby_blocks.zip

What is 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

# When “ntimes(23)” gets called, Proc object created
# 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
```

Lab:

Exercise 5: Ruby Closure
5512_ruby_blocks.zip

Code with Passion!
JPassion.com