Lab 5: Blocks, Procs, and Lambdas in Ruby

Learning Objectives:

1. Blocks in Ruby
2. Procs in Ruby
3. Introduction to Lambdas in Ruby
4. Introduction to Advanced Enumeration in Ruby

Supporting files:

1. Block_examples.zip
2. proc_examples.rb
3. lambdas_example.rb

1. Blocks in Ruby

A Ruby Block is an extremely powerful feature of the programming language, that allows to group a set of statements together. Ruby blocks can be defined with or without names, and they do not belong to any specific object. Blocks allow the code surrounded between braces or between do..end to be associated with method invocations, treated as parameters. Blocks often appear adjacently to a method and the code within the block is not executed as soon as encountered. A simple example of Ruby blocks is as follows:

```
10.times do
  puts "Hello from a block!"
end
```

In the above piece of code *times* is a method that executes the anonymous piece of code between *do* and *end*. Here one can see that the block is passed into the method times instead of any arguments. The following piece of code would also have the same output as the previous one.

```
10.times {puts "Hello from a block!"}
```

There are more methods like times, that take a block as an argument. For example, the each method as shown in the example below:

```
sum = 0

[1, 2, 3, 4, 5].each do |num|
  sum += num
end

######..or..######

 [1, 2, 3, 4, 5].each { |num| sum += num }

puts "Sum of the passed numbers: #{sum}"
```

The above example shows two different syntaxes for using the each method in Ruby.

The *yield* keyword

While the above examples show anonymous blocks, blocks with names are invoked from functions that have the same name as the block using the yield keyword.

```
def block_example
        puts “inside method”
        yield
        puts “back inside method again”
end
block_example {puts “inside block”}
```

Output:

```
        inside method
        inside block
        back inside method again
```

The yield keyword can be accompanied by parameters as well. The following example shows the procedure to that.

```
def block_test
        puts “inside method”
        yield 100
        puts “back inside method again”
end
block_test {|arg| puts “the yield has passed #{arg} to this block”}
```

Output:

```
        inside method
        the yield has passed 100 to this block
        back inside method again
```

---

2. Procs in Ruby

In Ruby everything that can be stored in a variable is an object. Procs in Ruby gives the ability to a block of code to be wrapped up in an object, store it in a variable or pass it to a method, and run the code in the block whenever needed (preferably more than once). So it is like a method itself, except that it is an object, and you we can store it or pass it around like any object. Here is an example of passing Ruby blocks into methods as Procs.

```
def block_test(&myblock)
        myblock.call
end
block_test { puts “Hello from the block”}
```

Output:

```
        Hello from the block
```

In the above example, since the last argument of our block_test method is preceded by &, we can pass our block_test block to this method. The passing ensures that this block will be assigned to the last parameter.

Procs in Ruby can be created and called independently as well, without being passed to any method. For example:

```
test_proc = Proc.new do
        puts “Hello from Proc”
end
test_proc.call
```

Output:

```
Hello from Proc
```

Procs can also take arguments in Ruby, for example:

```
test_proc = Proc.new do |something|
        puts "Let's try '+something'!"
end
test_proc.call 'Ruby programming'
```

Output:

```
Let's try Ruby programming
```

Methods with multiple procs as arguments

When we pass a proc into a method, we can control how, if, or how many times we call the proc. For example:

```
def test_method procArg
        procArg.call
end

testArg1 = Proc.new do
        puts "Hi there!"
end
testArg2 = Proc.new do
        puts "Bye bye!"
end
test_method testArg1
test_method testArg2
```

Output:

```
Hi there!
Bye bye!
```

Methods that return Procs

Methods not only can take procs as arguments but they can also return procs, here is an example of how to return a proc from a method:

```
def test procArg1, procArg2
        Proc.new do |arg|
                procArg2.call(procArg1.call(arg))
        end
end
square = Proc.new do |arg|
        arg * arg
end
double = Proc.new do |arg|
        arg + arg
end
squareAfterDouble = test double, square
```

```
puts squareAfterDouble.call (2)
test_method testArg2
```

Output:

```
16
```

3. Introduction to Lambdas in Ruby

Lambdas in Ruby are very similar to a Proc. However, the way Ruby handles both procs and lambdas are quite different. Such as,

1. Lamdbas check the number of parameters passed into the call, Procs do not.
2. Upon encountering the return keyword, Lamdbas return from the executing block but not from the lexically surrounding method call.

An example of these differences is as follows:

```
def test_lambda(arg)
        arg.call(1)
end
param_lambda = lambda {|a,b| puts "#{a} and #{b}"}
param_proc = Proc.new {|a,b| puts "#{a} and #{b}"}

test_lambda(param_proc)
test_lambda(param_lambda)
```

Output:

```
1 and
ArgumentError: wrong number of arguments (1 for 2)
```

4. Introduction to Advanced Enumeration in Ruby

In Ruby basic enumeration is done using blocks, the yield keyword, and the each method. In some advanced concepts of Ruby, *Lazy* enumeration is often considered a better way of processing a collection, as it will allow you to step through infinite sequences as far as you'd like to go.

Think of an assembly line of people making cakes where each person is responsible for only one step in the cake's creation. The first person bakes the cake, the next person adds the frosting, the last person adds the cherries. In this example, Ruby's lazy version of this is to have any number of orders of cake, but everyone takes the time to do just the first cake through every step of the process before continuing on to the next cake to make. If we don't use lazy enumeration, then each step would have to wait for the entire collection to be done one step at a time. For example, if we have 20 orders of cake, the person who bakes the cakes will have to bake 20 of them before any of them get their frosting or cherries added on by the next person. And each step in the line waits in a similar manner. Now, the bigger the collection we need to process, the more challenging it gets.

Creating a lazy enumerator in Ruby is as simple as calling lazy on an object with Enumerable included in it or to_enum.lazy on an object with each defined on it. For example,

```
(0..Float::INFINITY).take(2)
```

Output:

```
[0,1]
```

Lazy enumerations are among the more advanced concepts in Ruby. Lazy iterations can help greatly with splitting tasks off for different threads or background jobs to handle. The concept of lazy iteration offers the greatest flexibility. Some languages such as Rust with iterators, have made it their standard to be implemented lazily.

References:

[1] http://rubylearning.com/satishtalim/ruby_blocks.html
[2] https://www.tutorialspoint.com/ruby/ruby_blocks.htm
[3] http://ruby-for-beginners.rubymonstas.org/blocks/arguments.html
[4] http://radar.oreilly.com/2014/02/why-ruby-blocks-exist.html
[5] https://pine.fm/LearnToProgram/chap_10.html
[6] https://www.skorks.com/2009/08/more-advanced-ruby-method-arguments-hashes-and-blocks/
[7] https://blog.codeship.com/advanced-enumeration-with-ruby/
[8] http://www.tweetegy.com/2012/01/ruby-blocks-procs-and-lambdas/