Blocks

Blocks are very prevalent in Ruby, you can think of them as little anonymous functions that can be passed into methods.Blocks are enclosed in a do / end statement or between brackets {}, and they can have multiple arguments.

```
def forEach(num)
  num.times do |i|
    yield(i)
  end
end
forEach(5) { |i| puts "Call number #{i}" }

Call number 0
Call number 1
Call number 2
Call number 3
Call number 4
 => 5
```

So how does a method work with a block, and how can it know if a block is available? Well, to answer the first question, you need to use the yield keyword. When you use yield, the code inside the block will be executed.

```
def print_once
  yield
end

print_once { puts "In block" }

In block
 => nil


def print_twice
  yield
  yield
end

print_twice {  puts "In block" }
```

```
In block
In block
 => nil

You can call the yield with certain number of arguments:
def forEach(num)
  num.times do |i|
    yield(i)
  end
end
forEach(5) { |i| puts "Call number #{i}" }

Call number 0
Call number 1
Call number 2
Call number 3
Call number 4
 => 5
```

Blocks can also be explicit instead of implicit. You can name the block and pass it around if you need to.

```
def exp_block(&block)
  block.call # Same as yield
end

exp_block { puts "Explicit block" }

Explicit block
 => nil
```

Implicit Block invocation:

```
def sequence2(n, m, c)
i = 0
while(i < n)
 # loop n times
```

```
yield i*m + c
 # pass next element of the sequence to the block
i += 1
end
end
# Here is how you might use this version of the method
sequence2(5, 2, 2) {|x| puts x } # Print numbers 2, 4, 6, 8, 10

One of the features of blocks is their anonymity. They are not passed
to the method in a traditional sense, they have no name, and they are
invoked with a keyword rather than with a method.
```

Explicit Block Invocation:

```
def sequence3(n, m, c, &b) # Explicit argument to get block as a Proc
i = 0
while(i < n)
b.call(i*m + c)
 # Invoke the Proc with its call method
i += 1
end
end
# Note that the block is still passed outside of the parentheses
sequence3(5, 2, 2) {|x| puts x }
Notice that using the ampersand in this way changes only the method
definition
```

If you try to yield without a block you will get a no block given (yield) error.

Lambdas vs Procs

Blocks are syntactic structures in Ruby; they are not objects, and cannot be manipulated as objects. It is possible, however, to create an object that represents a block. Depending on how the object is created, it is called a proc or a lambda. Procs have block-like behavior and lambdas have method-like behavior. Both, however, are instances of class Proc. Procs and lambdas are objects, not methods, and they cannot be invoked in the same way that methods are. If p refers to a Proc object, you cannot invoke p as a method. But because p is an object, you can invoke a method of p

Procs and lambdas are a very similar concept, one of the differences is how you create them. A proc is the object form of a block, and it behaves like a block. A lambda has slightly modified behavior and behaves more like a method than a block. Calling a proc is like yielding to a block, whereas calling a lambda is like invoking a method.

In fact, there is no dedicated Lambda class. A lambda is just a special Proc object. Proc definition has instance methods lambda? to identify whether it is lambda or not.

```
my_lambda = -> { puts "This is a lambda" }
my_lambda.call

This is a lambda
 => nil
```

Lambdas can also take arguments:

```
times_two = ->(x) { x * 2 }
 => #<Proc:0x00000001764200@(irb):45 (lambda)>
times_two.call(10)
 => 20
```

If you pass the wrong number of arguments to a lambda, it will raise an exception, just like a regular method. But that’s not the case with procs.

```
t = Proc.new { |x,y| puts "I am Proc and I don't care about the given
arguments" }
 => #<Proc:0x000000016eff68@(irb):54>
t.call
I am Proc and I don't care about the given arguments
 => nil
```

Another difference between procs and lambdas is how they react to a return/break statement. A lambda will return normally, like a regular method. But a proc will try to return from the current context.

If you run the following code, you will notice how the proc raises a LocalJumpError exception. The reason is that you can't return from the top-level context.

```
# Should work
my_lambda = -> { return 1 }
puts "Lambda works and results is : #{my_lambda.call}"

Lambda works and results is : 1
 => nil


# Should raise exception
my_proc = Proc.new { return 1 }
puts "Proc donot work and result is: #{my_proc.call}"

LocalJumpError: unexpected return
      from (irb):62:in `block in irb_binding'
      from (irb):63
```

If the proc was inside a method, then calling return would be equivalent to returning from that method.

```
def proc
  puts "Before returning from proc"
  my_proc = Proc.new { return 2 }
  my_proc.call
  puts "After returning from proc" # Never get executed
end

p proc

Before returning from proc
```

```
2
 => 2


def lamda
  puts "Before returning from lamda"
  my_lamda = -> { return 2 }
  my_lamda.call
  puts "After returning from lamda"
end

p lamda

Before returning from lamda
After returning from lamda
nil
 => nil
```

Summary of how procs and lambdas are different:

- Lambdas are defined with -> {} and procs with Proc.new {}.
- Procs return from the current method, while lambdas return from the lambda itself.
- Procs don't care about the correct number of arguments, while lambdas will raise an exception.