**Blocks, Procs and Lambda**

## **Blocks**

* Anonymous functions
* Implicitly returns

**How a normal function looks like :**

def add(num1, num2)

num1 + num2

end

result = add(5, 10)

puts result

**How a block looks like :**

[1, 2, 3].each {|num| puts num \* 2}

**Defining blocks => 2 ways**

* Single line blocks**(Uses {})**

[1, 2, 3].each {|num| puts num \* 2}

* Multi line blocks**(Uses do..end)**

[1, 2, 3].each do |ele|

puts ele \* 2

end

**Explicitly Returning a block**

[1, 2, 3].each do |ele|

puts ele \* 3

return ele

end

**Yield**

* Transfers control from a method to a block that is passed to that method

def display

puts "Inside a function"

yield

puts "Came back to function"

end

display {puts "Inside a block"}

* Can use multiple yields inside a method

def display

puts "Inside a function"

yield

yield

puts "Came back to function"

end

display {puts "Inside a block"}

* Throws error when there is no block

def display

puts "Inside a function"

yield

puts "Came back to function"

end

display =>throws error

* To handle that , use **block\_given? ,** a predefined method to check if a block is present

def display

puts "Inside a function"

yield

if !block\_given?

puts "No block"

end

puts "Came back to function"

end

display

* Yield with arguments

def add

yield(5)

end

add {|num1, num2 = 10| puts num1 + num2}

* Yield with return value

def yield\_with\_return\_value

nums = yield

puts nums

end

yield\_with\_return\_value do

100

end

## **Procs**

* A Proc is essentially a block of code that is wrapped inside an object.
* Unlike a typical block, a Proc can be assigned to variables, passed to methods, and executed at any time.

**Creating a proc**

* Using **Proc.new**

my\_proc = Proc.new { |x| puts x \* 2 }

* Using **proc** keyword

my\_proc = proc { |x| puts x \* 3 }

**Calling a proc**

square = Proc.new {|x| x\*\*2 }

* Using call method

puts square.call(3)

* Using .()

puts square.(3)

* Using []

puts square[3]

**Assigning a proc to a variable**

my\_proc = Proc.new { |x| puts x \* 2 }

**Passing a proc to a method**

def greet\_user(greeting\_proc)

greeting\_proc.call

end

greeting = Proc.new { puts "Good morning!" }

greet\_user(greeting)

**Returning a proc**

def return\_proc

Proc.new { puts "Proc returned from a method!" }

end

my\_proc = return\_proc

my\_proc.call

**Using multiple blocks via Procs**

double = Proc.new { |x| puts x \* 2 }

triple = Proc.new { |x| puts x \* 3 }

procs = [double, triple]

def apply\_procs(procs)

procs.each do |proc|

proc.call(7)

end

end

apply\_procs(procs)

**Converting Blocks to procs**

def run\_proc(&block)

block.call

end

run\_proc { puts "Block converted to Proc!" }

**Passing Procs as objects**

class GreaterThan

def initialize(threshold)

@threshold = threshold

end

def to\_proc

proc { |x| x > @threshold }

end

end

greater\_than\_3 = GreaterThan.new(3)

result = [1, 2, 3, 4, 5].select(&greater\_than\_3)

puts result.inspect

**Procs are closures**

def create\_counter

count = 0

Proc.new do

count += 1

count

end

end

counter = create\_counter

puts counter.call

puts counter.call

puts counter.call

**Comparing Procs**

proc1 = Proc.new { puts "Proc 1" }

proc2 = proc1

proc3 = Proc.new { puts "Proc 1" }

puts proc1 == proc2

puts proc1 == proc3

**Lambda**

* Lambda is a type of Proc that is more strict in terms of argument handling.

**Creating a lambda**

* Using **lambda** keyword

my\_lambda = lambda { |x| puts "Hello, #{x}!" }

* Using **->** operator

my\_lambda = ->(x) { puts "Hello, #{x}!" }

**Calling a lambda**

* Using **call** method

my\_lambda.call("John")

* Using **.()**

puts my\_lambda.(5)

**Arguments passed to a Lambda**

my\_lambda = lambda { |x| x \* 2 }

my\_lambda.call(5)

my\_lambda.call

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Blocks** | **Procs** | **Lambdas** |
| **Definition** | A block is a chunk of code enclosed in either braces {} or do...end. | A Proc is an object that encapsulates a block of code. | A lambda is a special type of Proc with stricter rules. |
| **Creation** | Defined inline with a method call. | Created using Proc.new or the proc method. | Created using lambda or the -> syntax. |
| **Return Behavior** | Returns from the context of the method that called it. | Returns from the enclosing method | Returns same as a block |
| **Argument Handling** | Can accept a variable number of arguments, but ignores extra ones without error. | Can accept a variable number of arguments but will ignore extra ones without error. | Requires the exact number of arguments; raises an error if the number is incorrect. |
| **Arity** | Does not enforce arity; extra arguments are ignored. | Does not enforce arity; extra arguments are ignored. | Enforces arity; raises an ArgumentError if the number of arguments is incorrect. |
| **Scope** | Has access to the local variables from the method where it's defined. | Has access to the local variables from the scope in which it was created. | Has access to the local variables from the scope in which it was created. |
| **Usage Context** | Typically used for iteration or passing chunks of code to methods. | Often used for callbacks, event handlers, or storing reusable code. | Typically used for more complex functional programming where strict argument handling is required. |