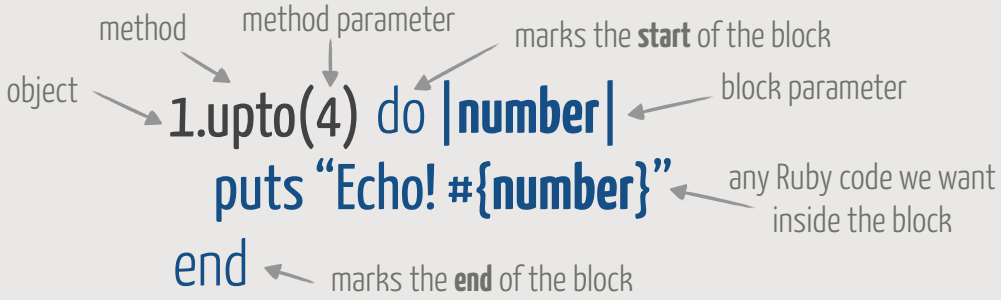
Multi-line Block:



Single-line Block:

Scoping Rules

```
name = "Curly"
number = 99
temp = 32.0

3.times do |number; temp|
  name = "Moe"
  age = 25
  temp = 98.6
  puts "#{number}: #{name} is #{age} (#{temp})"
end
```

variables outside the block can be shared inside the block

block parameters are always local to the block

block-level variables are reserved for inside the block

variables defined inside a block are local to the block

```
0: Moe is 25 (98.6)!
1: Moe is 25 (98.6)!
2: Moe is 25 (98.6)!
```

puts age	Error
puts name	Moe
puts number	99
puts temp	32.0

each Iterator

orders = [moe@example.com $10.00 , larry@example.com $20.00 , curly@example.com $30.00]

curly@example.com $30.00
larry@example.com $20.00
moe@example.com $10.00

one-by-one each element in the array is assigned to the block parameter

acts like a loop

```
orders.each do |order|
  puts order.email
end
```


scores = [85, 105, 71, 113, 122, 94]

select

synonym: find_all

```
high_scores = scores.select { |score| score > 100 }
```

high_scores == [105, 113, 122] ← returns an array containing all elements for which the block is **true**

reject

```
low_scores = scores.reject { |score| score > 100 }
```

low_scores == [85, 71, 94] ← returns an array containing all elements for which the block is **false**

scores = [85, 105, 71, 113, 122, 94]

any?

```
high_scores = scores.any? { |score| score > 100 }
high_scores == true
```

If the block returns a **true** value, then the method returns true.

detect

```
first_high_scores = scores.detect { |score| score > 100 }
first_high_scores == 105
```

Returns the **first** element in the collection that matches the criteria in the block.

scores = [85, 105, 71, 113, 122, 94]

partition

name of **first** array — name of **second** array

```
high, low = scores.partition { |score| score > 100 }
```

an array containing all elements for which the block is **true**

an array containing all elements for which the block is **false**

```
high == [ 105, 113, 122 ]
low == [ 85, 71, 94 ]
```

map

```
scores = [85, 105, 71, 113, 122, 94]
```

synonym: collect

using **map!** will modify the original array

```
doubled = scores.map { |score| score * 2}
doubled == [170, 210, 142, 226, 244, 188]
```

returns an array the **same size** as the original array

reduce

synonym: inject

initial value of sum

```
total = scores.reduce(0) { |sum, score| sum + score }
total == 590
```

the accumulator value

yielding

yields control to the associated block

```
def roll
  number = rand(1..6)
  result = yield("Larry", number)
  puts "The block returned #{result}."
end
```

variable that captures the value of yield returned from the block

parentheses are optional

arguments to yield are passed directly to the associated block as block parameters

```
roll do |name, number|
  puts "#{name} rolled a #{number}!"
  number * 10
end
```

the **value of the last expression** in the block is automatically passed back to the method as the **value of yield**

custom iterators

Defining an **each** method and **including Enumerable** allows you to call all the Enumerable methods outside the class. For example:

queue.select { |m| m.duration > 140 }

```
class MovieQueue
  include Enumerable

  def initialize(name)
    @name = name
    @movies = []
  end

  def add_movie(movie)
    @movies << movie
  end

  def each
    @movies.each { |movie| yield movie}
  end
end
```

Defining an each method that iterates through all the elements in the array allows you to call **each** outside the class, like so:

queue.each { |movie| movie.title }

each method on Array class

Execute Around

This pattern is used when you want to **execute around some code** like a toggle switch: turn something on, yield to a block to do something, and then remember to turn it off.

```
def in_airplane_mode
  @airplane_mode = true
  yield
rescue Exception => e
  puts e.message
ensure
  @airplane_mode = false
end
```

captures any exception and prints it out

guarantees that any code in this clause is always run regardless of whether an exception is raised

Self Yield

This pattern, sometimes referred to as **Self Yield** or **Block Initialization**, makes it clear that all the code in the block is focused on initializing the object.


```
class Canvas
  attr_accessor :width, :height, :color

  def initialize
    @width = 100
    @height = 100
    @color = :green
    yield self if block_given?
  end
end
```

yields the current object to the block for further initialization, but only if a block is given

references the current object being created


```
canvas = Canvas.new do |c|
  c.width = 800
  c.height = 600
end
```


Managing Resources

This pattern is commonly used when dealing with expensive and/or limited resources such as network connections, files, and such. It ensures they're opened and closed consistently.

```
def self.open(user, password)
  gateway = self.new(user, password)
  gateway.connect

  return gateway unless block_given?

  begin
    yield(gateway)
  ensure
    gateway.disconnect
  end
end

Gateway.open("admin", "secret") do |g|
  # use gateway referenced in 'g'
end
```

open resource

yield resource

ensure resource is closed

```
def self.open(user, password)
  resource = self.new(user, password)
  resource.open

  return resource unless block_given?

  begin
    yield(resource)
  ensure
    resource.close
  end
end

Resource.open("admin", "secret") do |r|
  # use the resource referenced in 'r'
end
```