

Ruby: Blocks, Hashes, and Symbols

Computer Science and Engineering ■ College of Engineering ■ The Ohio State University

Lecture 8

Blocks

- A block is a *statement(s)* passed in as an *argument* to a function

```
5.times do
  puts 'hello world'
end
```

- Equivalent, but more succinct:

```
5.times { puts 'hello world' }
```

- A block can, itself, have parameters!

```
5.times { |n| puts n**2 }
```

- Method calls block, passing in arguments

Calling Blocks

- Within a function, the passed-in block is called with keyword "yield"

```
def fib_up_to(max)
  i1, i2 = 1, 1
  while i1 <= max
    yield i1 if block_given?
    i1, i2 = i2, i1 + i2
  end
end

fib_up_to(1000) { |f| print "#{f} " }
fib_up_to(1000) { |f| sum += f }
```

Idioms for Blocks

- Bracketed code (eg open, do stuff, close)

```
File.open('notes.txt', 'w') do |file|  
  file << 'work on 3901 project'  
end # file closed by open method
```
- Nested scope (eg for initialization code)

```
agent = Mechanize.new do |a|  
  a.log = Logger.new ('log.txt')  
  a.user_agent_alias = 'Mac Safari'  
end # isolates init'n code and temp var a
```
- Iteration (very common)...

Simple Iteration

- While/until loop: Boolean condition

```
while boolean_condition
```

```
...
```

```
end
```

- For-in loop: iterate over arrays (and other things like ranges)

```
for var in array
```

```
...
```

```
end
```

- Example

```
for str in 'hi'..'yo'
```

```
  puts str.upcase
```

```
end
```

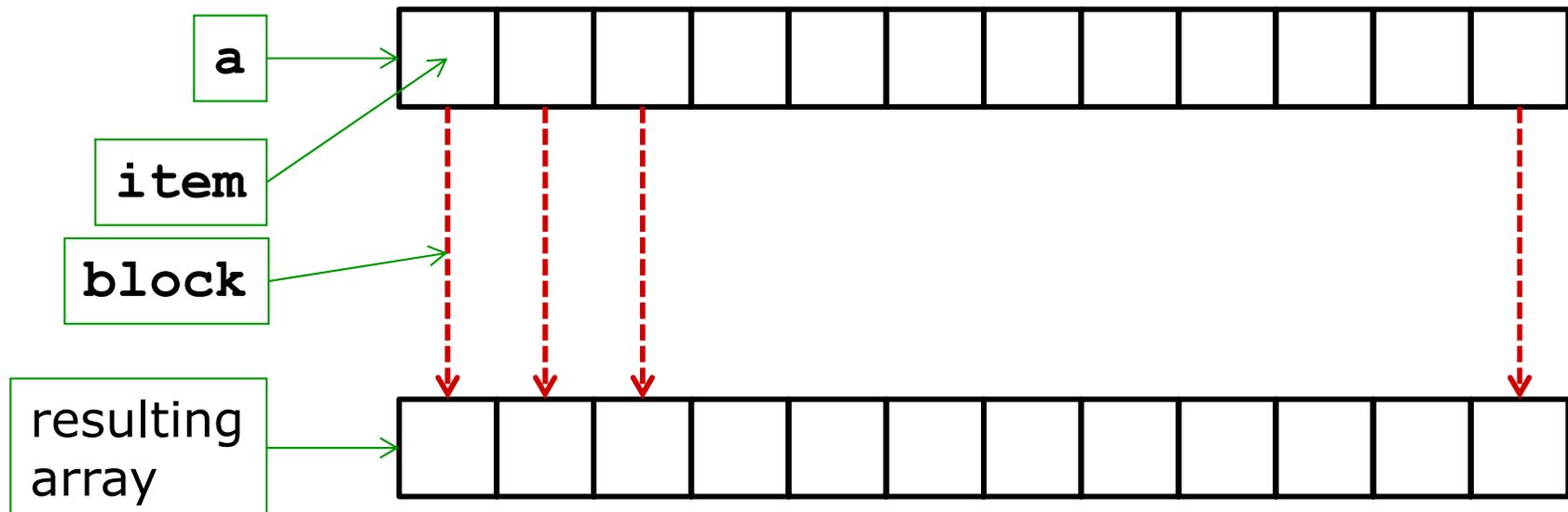
- Usually avoided (rubystyle.guide/#no-for-loops)

Iterating on Arrays Using Blocks

- Do something with every element
`a.each { |str| puts str.upcase }`
- Do something with every index
`a.each_index { |i| print "#{i}--" }`
- Fill array with computed values
`a.fill { |i| i * i }`
`a.fill { |i| [] } # or omit i: { |_| [] }`
- Search
`a.index { |x| x > limit }`
- Filter
`a.select! { |v| v =~ /[aeiou]/ }`
`a.reject! { |v| v =~ /[aeiou]/ } # aka filter`
- Sort
`a.sort! { |x, y| x.length <=> y.length }`

Map

- Transform an array into a new array, *element by element*
- Uses *block* to calculate each new value
`a.map { |item| block } # also !`



Map: Examples

```
names = %w{ali noah marco xi}
```

```
  #=> ["ali", "noah", "marco", "xi"]
```

```
names.map { |name| name.capitalize }
```

```
  #=> ["Ali", "Noah", "Marco", "Xi"]
```

```
names.map { |name| name.length }
```

```
  #=> [3, 4, 5, 2]
```

```
[1, 2, 3, 4].map { |i| i**2 }
```

```
  #=> [1, 4, 9, 16]
```

```
[1, 2, 3, 4].map { |i| "x^{i}" }
```

```
  #=> ["x^1", "x^2", "x^3", "x^4"]
```

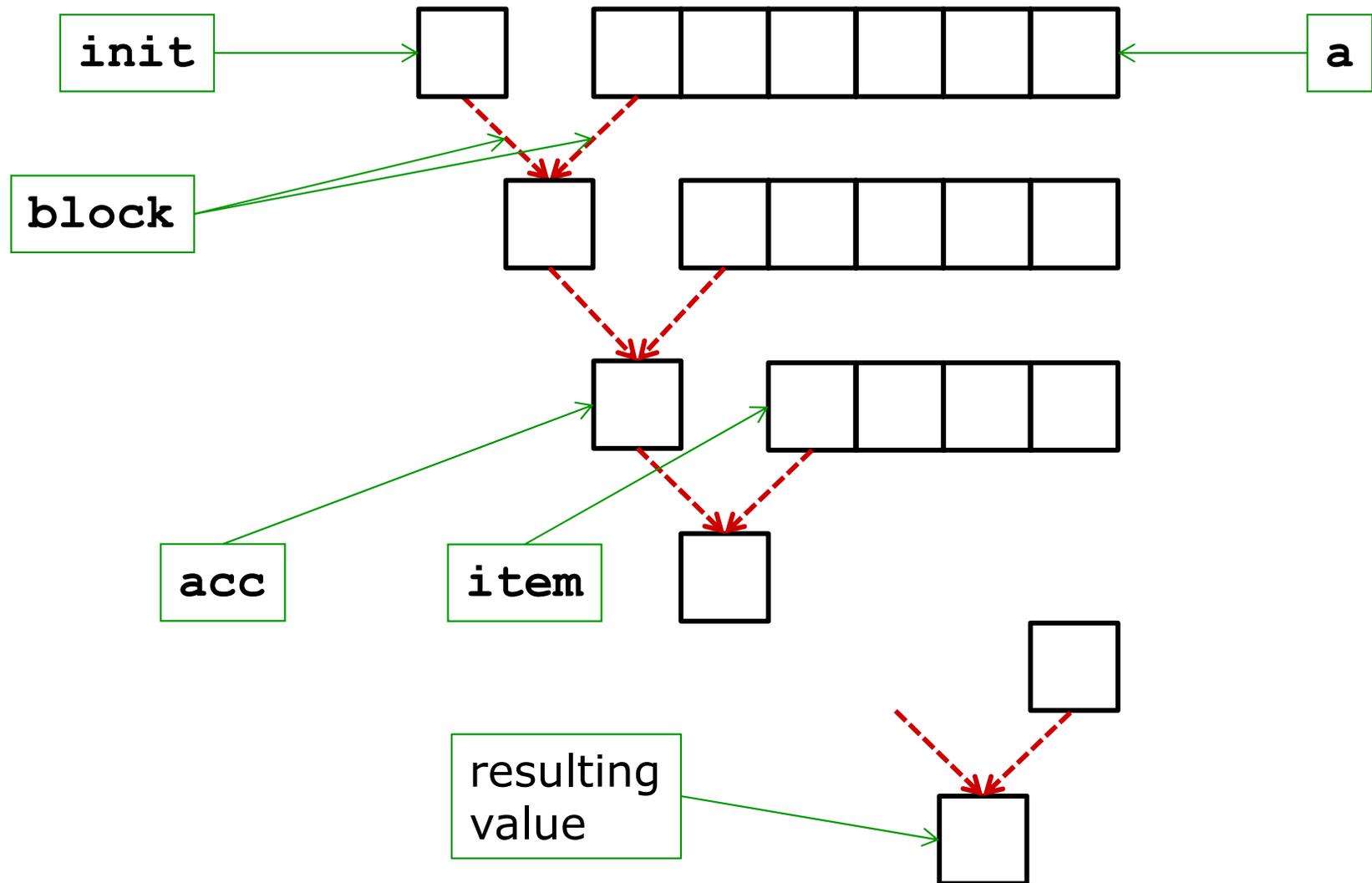
Reduce

- Transform an array into a *single value*, by incorporating one element at a time
 - Also called “fold”, or “inject”
- Uses block with 2 arguments: current accumulation and next array element

```
a.reduce(init) { |acc, item| block }
```

 - Value *returned* by block is the next `acc`
 - `a[0]` is initial `acc`, if `init` not provided
- Example: Sum the values of an array
 - `[15, 10, 8] → 0 + 15 + 10 + 8 → 33`

Reduction Chain



Reduce: Examples

```
[3, 4, 5].reduce { |sum, i| sum + i }      #=> 12
```

```
[1, 2, 3, 4, 5].reduce '' do |str, i|  
  str + i.to_s  
end                                         #=> "12345"
```

```
words = %w{cat sheep bear}  
words.reduce do |memo, word|  
  memo.length > word.length ? memo : word  
end                                         #=> "sheep"
```

```
[1, 2, 3].reduce [] do |acc, i|  
  acc.unshift i  
end                                         #=> ???
```

Module: Enumerable

□ Quantify over elements

```
['hi', 'yo!'].all? { |w| w.length > 2 }  
(0..100).any? { |x| x < 0 } #=> false  
[1, 2, 3].none? { |x| x % 2 == 0 }
```

□ Min/Max

```
words.max_by { |x| x.length }
```

□ Search

```
(1..10).find_all { |i| i % 3 == 0 }  
      #=> [3, 6, 9]
```

□ Map/reduce (only non-! version)

```
(5..8).map { 2 } #=> [2, 2, 2, 2]  
(1..10).reduce(+:) #=> 55
```

```
book.reduce(0) { |sum, w| sum + w.length }
```

Your Turn

- Given a string
- Produce an array of indices where '#' occurs in the string
- Example:
 - Given
`'a#asg#sdfg#d##'`
 - Result
`[1, 5, 10, 12, 13]`

Your Turn

- Given an array of integers
- Produce the array that includes only the even elements, each squared
- Example:
 - Given
[1, 2, 3, 7, 7, 1, 4, 5, 6, 2]
 - Result
[4, 16, 36, 4]

Your Turn

- Given an array of (a mix of) integers and array of integers, where the (top level) integers are unique
- Remove from the contained arrays all occurrences of the top level integers
- Example:
 - Given
[3, 5, [4, 5, 9], 1, [1, 2, 3, 8, 9]]
 - Result
[3, 5, [4, 9], 1, [2, 8, 9]]

Example: What Does This Do?

```
words = File.open('tomsawyer.txt') { |f|
  f.read }.split

freq, max = [], ''
words.each do |w|
  max = w if w.length > max.length
  freq[w.length] = 0 if !freq[w.length]
  freq[w.length] += 1
end

puts words.length
puts words.reduce(0) { |s, w| s + w.length }
freq.each_index do |i|
  puts "#{i}-letter words #{freq[i]}"
end

puts max
```

Hashes

□ Partial map: keys \rightarrow values

■ Keys must be unique

□ Indexed with array syntax []

```
h['hello'] = 5
```

□ Literal syntax for initialization

```
h = { 'red'    => 0xf00 ,  
      'green' => 0x0f0 ,  
      'blue'  => 0x00f  }
```

□ Optional: Instantiate with a default value (or block)

```
h1 = Hash.new 0 #=> beware aliases
```

```
h2 = Hash.new { |h, k| h[k] = k + k }
```

Using Hashes

```
h = { 'age' => 21 }    # create new Hash
h['age'] += 1         # mutable values
h['id'] = 0x2a        # can grow
h.size               #=> 2
h['name'] = 'Luke'   # heterog. values
h[4.3] = [1, 3, 5]  # heterog. keys
h.delete 'id'        # can shrink

# h == { 'age' => 22,
#       'name' => 'Luke',
#       4.3 => [1, 3, 5] }
```

Example

```
list = %w{cake bake cookie car apple}
```

```
# Group by string length:
```

```
groups = Hash.new{ |h, k| h[k] = [] }
```

```
list.each { |v|
```

```
  groups[v.length] << v
```

```
}
```

```
# groups == { 4 => ["cake", "bake"],
```

```
#       6 => ["cookie"],
```

```
#       3 => ["car"], 5 => ["apple"] }
```

Your Turn

- Write the Ruby code that, given an array of strings, computes frequency of occurrence of each word
- Example:
 - Given
`["car", "van", "car", "car"]`
 - Compute
`{"car" => 3, "van" => 1}`

Example

```
list = %w{car van car car}
```

```
# Your code here
```

```
groups #=> {"car" => 3, "van" => 1}
```

Using Blocks with Hashes

- Do something with every key/value pair

```
h.each { |k, v| print "(#{k},#{v})" }
```

- Do something with every key or value

```
h.each_key { |k| print "#{k}--" }
```

```
h.each_value { |v| print "#{v}--" }
```

- Combine two hashes

```
h1.merge(h2) { |k, v1, v2| v2 - v1 }
```

- Filter

```
a.delete_if { |k, v| v =~ /[aeiou]/ }
```

```
a.keep_if { |k, v| v =~ /[aeiou]/ }
```

Immutability of Keys

- Rule: Once a key is in a hash, *never* change its value

```
grades[student] = 'C+'  
student.wake_up! # danger
```

- Problem: Aliases

- “Solution”: For strings, Ruby *copies* (and *freezes*) a string when added to a hash

```
a, b = String.new('fs'), String.new('sn')  
h = {a => 34, b => 44}  
puts a.object_id, b.object_id  
h.each_key { |key| puts key.object_id }
```

Symbols

- Roughly: *unique & immutable* strings
- Syntax: prefix with ":"
 - `:height`
 - `:'some symbol'`
 - `"#{name}'s crazy idea"`
- Easy (too easy?) to convert between symbols and strings
 - `:name.to_s` $\#=>$ `"name"`
 - `'name'.to_sym` $\#=>$ `:name`
- But symbols are *not* strings
 - `:name == 'name'` $\#=>$ `false`

Operational View

- A symbol is created once, and all uses refer to that same object (aliases)
- Symbols are immutable
- Example

```
[].object_id ==> 200
```

```
[].object_id ==> 220
```

```
[].equal? [] ==> false
```

```
:world.object_id ==> 459528
```

```
:world.object_id ==> 459528
```

```
:world.equal? :world ==> true
```

Symbols as Hash Keys

- Literal notation, but note colon location!

```
colors = {red: 0xf00,  
          green: 0x0f0,  
          blue: 0x00f}
```
- This is just syntactic sugar
 - `{name: value}` same as `{:name => value}`
 - The key is a *symbol* (eg `:red`)
- Pitfalls

```
colors.red      #=> NoMethodError  
colors["red"]  #=> nil  
colors[:red]   #=> 3840 (ie 0xf00)
```

Keyword Arguments

- Alternative to positional matching of arguments with formal parameters

```
def display(first:, last:)  
  puts "Hello #{first} #{last}"  
end  
  
display first: 'Mork', last: 'Ork'  
display last: 'Hawking', first: 'Steven'
```

- Providing a default value makes that argument optional

```
def greet(title: 'Dr.', name:)  
  puts "Hello #{title} #{name}"  
end
```

- Benefits: Client code is easier to read, and flexibility in optional arguments

Summary

- Blocks
 - Code passed as argument to a function
 - Elegant iteration over arrays
- Enumerable
 - Many useful iteration methods
- Hashes
 - Partial maps (aka associative arrays)
- Symbols
 - Unique, immutable strings
 - Often used as keys in hashes