

Blocks

- Let's write a simple for loop to search through an array looking for a particular value:

```
$flock = ["huey", "dewey", "louie"]

def isDuck?(name)
  for i in 0..$flock.length
    if $flock[i] == name then
      return true
    end
  end
  return false
end

puts isDuck?("dewey"), isDuck?("donald")
```

CSc 372

Comparative Programming Languages

29 : Ruby — Blocks

Department of Computer Science
University of Arizona

collberg@gmail.com

Copyright © 2011 Christian Collberg

Christian Collberg

Iterators

- Ruby's *iterators* are an easier way to do this.
- The `Array` class implements a method `find` that iterates through the array.

```
def isDuck?(name)
  $flock.find do |x|
    x == name
  end
end

puts isDuck?("dewey")
puts isDuck?("donald")
```

Yield

- A block is enclosed within `{}` or `do...end`. Arguments to the block (there can be more than one) are given within `|...|`.
- A block is passed to a method by giving it after the list of "normal" parameters.
- The method invokes the block by using `yield`.
- `yield` can take an argument which the method passed back to the block.

Yield...

```
def triplets()
  yield "huey"
  yield "dewey"
  yield "louie"
end

triplets() {|d| puts d}

triplets() do |d|
  puts d
end
```

Factorial

- Here's the factorial function, as an iterator.

```
def fac(n)
  f = 1
  for i in 1..n
    f *= i
    yield f
  end
end

fac(5) {|f| puts f}
```

Passing arguments

- yield can pass more than one value to the block.

```
def fac(n)
  f = 1
  for i in 1..n
    f *= i
    yield i, f
  end
end

fac(5) do |i, x|
  puts "#{i}! = #{x}"
end
```

Nesting iterators

- Iterators can be nested.

```
fac(3) do |i, x|
  fac(3) do |j, y|
    puts "#{i}! * #{j}! = #{x*y}"
  end
end
```

Scope

- A local variable which is active when the block is started up, can be accessed (and modified) within the block.

```
def sumfac(n)
  y = 0
  fac(n) do |i,x|
    y = y + x
  end
  return y
end

puts sumfac(5)
```

Implementing Array#find

- We can implement our own find method:

```
def find(arr)
  for i in 0..arr.length
    if yield arr[i] then return true end
  end
  return false
end

puts find($flock) {|x| x=="dewey"}
puts find($flock) {|x| x=="donald"}
```

Array#collect

- collect applies the block to every element of an array, creating a new array. This is similar to Haskell's map.

```
$flock = ["huey","dewey","louie"]
$flock.each {|x| puts x}

puts $flock.collect {|x| x.length}
puts $flock.collect do |x|
  "junior woodchuck, General " + x
end
```

Array#inject

- inject(init) is similar to Haskell's foldl.
- inject() without an argument is like Haskell's foldl1, i.e. it uses the first element of the array as the starting value.

```
x = $flock.inject("") do |elmt,total|
  total = elmt + " " + total
end
puts x

x = $flock.inject() do |elmt,total|
  total = elmt + " " + total
end
puts x
```

Exercise — MyHash

- Let's write our own version of Ruby's Hash class, called MyHash.
- The hash table should be implemented as an array of buckets $[0..size-1]$, where each bucket i is an array of $[key, value]$ pairs and such as

$$i = \text{key.hash mod size}$$

- First, declare the class and add a constructor.
- The constructor should take one argument, the size (number of buckets). It should create the buckets (an array of nil values) and set an instance variable `@size` to the number of buckets.
- HINT: `Array.new(size=...,obj=...)` creates an array of size `size`, with each value being `obj`.

Exercise — MyHash — get

- Now implement the `get(key)` method.
- The algorithm is as follows:
 - 1 Compute the bucket number for the key.
 - 2 Check if the bucket is empty (nil). If it is, return nil.
 - 3 Look through the table to see if there's an element in the bucket with the right key. If so, return the value. Otherwise, return nil.

Exercise — MyHash — put

- Now implement the `put(key, value)` method.
- The algorithm is as follows:
 - 1 Compute the bucket number for the key, i.e. `key.hash() mod` the size of the bucket array.
 - 2 Check if the bucket is empty (nil). If so, set it to be an empty list.
 - 3 Look through the table to see if there's already an element in the bucket with the right key. If so, change the element to the new value. Otherwise, add the `[key,value]` pair to the end of the bucket.
- HINT: `array.map! { |item| block }` invokes the block once for each element of self, replacing the element with the value returned by block.

Exercise — MyHash...

This code

```
h = MyHash.new(10)
h.put("hey", "there")
h.put("yo", "dude")
puts h.get("hey")
puts h.get("yo")
h.put("hey", "baby")
puts h.get("yo")
puts h.get("hey")
```

should generate this output:

```
there
dude
dude
baby
```

Exercise — MyHash — each

- Now implement the `each` method which yields each element at a time.
- Use `each` to implement `keys()` and `values()` methods that yields each element at a time.
- Extend `keys()` such that it can yield each element at a time (if you pass it a block) or returns an array of keys if you don't.
- HINT: The method `block_given?` returns true if you've passed a block to the method.
- Add a method `to_s()` that return the key-value pairs of the hashtable as a string.

Exercise — MyHash — Example. . .

```
puts h.to_s()
```

should print

```
hey => baby  
yo => dude
```

Exercise — MyHash — Example. . .

```
h.keys() {|x| puts x}  
puts "-----"  
s = h.keys()  
puts s
```

should print

```
hey  
yo  
-----  
hey  
yo
```

Exercise — MyHash — Example. . .

- Extend the class so that in addition to using `put` and `get` you can also use `[]=` and `[]`. Example:

```
h["banana"] = "fruit"  
puts h["banana"]
```

should print

```
fruit
```
- HINT: `alias :newmethod :oldmethod` makes a new method `newmethod` that simply calls `oldmethod`.

Exam Problem I — 372 Fall 2008

Let's implement methods `map`, `filter`, and `foldr`, corresponding to their Haskell namesakes, but this time in Ruby! Here is the class definition:

```
class Array
  def Array.map(a)
    ...
  end

  def Array.filter(a)
    ...
  end

  def Array.foldr(a,z)
    ...
  end
end
```

Exam Problem I — 372 Fall 2008

Each method is passed an array `a` as input and returns a new array as output. In Haskell these higher-order functions would also be passed a function as argument, but here in Ruby they're instead passed a block. The `foldr` method also has an argument `z`, the starting value.

Exam Problem I(a) — 372 Fall 2008

Write the `Array.map` method. This example

```
a = Array.map([1,2,3]) do |x|
  x+1
end
puts a
```

should print out

```
2
3
4
```

Exam Problem I(b) — 372 Fall 2008

Write the `Array.filter` method. This example

```
a = Array.filter([1,2,3,4,5]) do |x|
  x % 2 == 0
end
puts a
```

should print out

```
2
4
```

Exam Problem I(c) — 372 Fall 2008

Readings

Write the `Array.foldr` method. These examples

```
puts Array.foldr([1,2,3,4,5],0) do |x,z|
  x+z
end
puts Array.foldr([1,2,3,4,5],0) do |x,z|
  x-z
end
puts Array.foldr(["aaa","bbb","ccc"],"") do |x,z|
  x+z
end
puts a
should print out
15
3
aaabbbccc
```

- Read Chapter 4, page 49–55, in *Programming Ruby — The Pragmatic Programmers Guide*, by Dave Thomas.
- Here's the documentation for the Array class:

<http://www.ruby-doc.org/core/classes/Array.html>

Yum!



烤鸭
Kào Yā