



CSE 413 Spring 2011

# Ruby Blocks, Procs, and Closures

# Blocks

- Any method call can be followed by a block. The block is executed by the method – when depends on the method

```
words = [ "fee", "fie", "foe", "fum" ]
```

```
words.each { | w | puts w }
```

```
all_words = ""
```

```
words.each { | w | all_words = all_words + w + " " }
```

# Block Execution

- A block is executed in the context of the method call
  - Block has access to variables at the call location
  - Return in a block returns from surrounding method(!)

```
def search(it, words)
  words.each { | w | if it == w return }
  puts "not found"
end
```

# yield

- Any method call can be followed by a trailing block. A method “calls” the block with a yield statement.

```
def repeat
  yield
  yield
end
repeat { puts "hello" }
```

Output:

```
hello
hello
```

# yield with arguments

- If the block has parameters, use expressions with yield to pass arguments

```
def xvii
  yield 17
end
xvii { | n | puts n+1 }
```

- This is exactly what an iterator does

# Blocks and Procs

- Blocks (and methods) are not objects in Ruby
  - i.e., not things that can be passed around as first-class values
- But we can create a Proc object from a block
  - Procs are real closures consisting of the block and the surrounding environment
  - Variations: procs and lambdas; slightly different behavior
  - Several different ways to construct these; see the language documentation for details

# Making Procs

- A method can have a parameter that explicitly represents the block

```
def return_a_block (& block)
  block.call(17)
  return block
end
```

- The ‘&’ turns the block into a proc object
- Proc objects support a “call” method

# Proc.new; lambdas

- Can also create a proc object explicitly

```
p = Proc.new { | x, y | x+y }
```

```
...
```

```
p.call(x,y)
```

- The kernel's lambda method also creates proc objects

```
is_positive = lambda { |x| x > 0 }
```



# Procs vs. Lambdas

- A Proc is a block wrapped in an object – and behaves just like a block
  - In particular, a return in a Proc will return from the *surrounding* method where the Proc's closure was created
    - Error if that method has already terminated
- A Lambda is more like a method
  - Return just exits from the lambda

# Functional Programming in Ruby

- Ruby is not a functional programming language, but with blocks, procs, and lambdas, you can do pretty much anything you could in a functional language
- Big difference is that Ruby is object-oriented, meaning dynamic dispatch, classes, inheritance, etc.
  - More to come on that...
- Reference on Ruby blocks, etc.: *The Ruby Programming Language*, ch. 6; Flanagan & Matsumoto