CMSC 330
Organization of Programming Languages

Code Blocks

Code Blocks

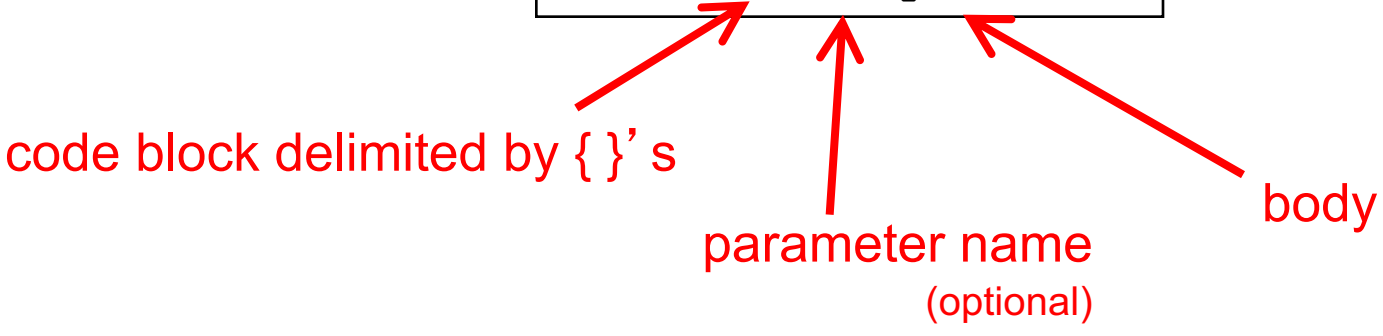
- A code block is a piece of code that is invoked by another piece of code

- Code blocks are useful for encapsulating repetitive computations

Array Iteration with Code Blocks

- The Array class has an each method
  - Takes a code block as an argument

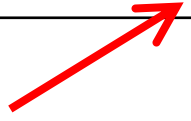
```
a = [1,2,3,4,5]
a.each { |x| puts x }
```



Array Iteration with Code Blocks

- The Array class has an each method
  - Takes a code block as an argument

```
a = [1,2,3,4,5]
a.each do |x| puts x end
```



or delimited by do ... end

So, What *Are* Code Blocks?

- A code block is like a special kind of method

```
{ |y| x = y + 1; puts x }
```

is almost the same as

```
def m(y) x = y + 1; puts x end
```

- The each method invokes the given code block
  - This is called higher-order programming
    - In other words, methods take other (almost-)methods as arguments

Quiz 1: What is the output?

```
a = [1,2,3,4]
sum = 0
a.each { |x| sum += 2*x }
puts sum
```

A. 10

B. 30

C. 20

D. 0

Quiz 1: What is the output?

```
a = [1,2,3,4]
sum = 0
a.each { |x| sum += 2*x }
puts sum
```

A. 10

B. 30

C. **20**

D. 0

More Code Blocks for Arrays

- Code block in **`each`** does not modify array

```
a = [1,2]
a.each { |x| x = x*x }
puts a[1]
# outputs 2, not 4
```

- **`a.find`** returns first element of **`a`** for which the block returns true

```
[1,2,3,4,5].find { |y| y % 2 == 0 }
[5,4,3].collect { |x| -x }
```

- **`a.collect`** applies block to each element of **`a`** and returns new array; **`collect!`** modifies **`a`**

Quiz 2: What is the output

```
a = [20,15,10,5]
a.collect! { |x| x*x }
puts a[1]
```

A. 10
B. 15
C. 225
D. 400

Quiz 2: What is the output

```
a = [20,15,10,5]
a.collect! { |x| x*x }
puts a[1]
```

A. 10

B. 15

C. **225**

D. 400

Code Blocks for Numbers, Strings

```
3.times { puts "hello"; puts "goodbye" }
5.upto(10) { |x| puts(x + 1) }
```

- n.times runs code block n times
- n.upto(m) runs code block for integers n..m

```
s = "Student,Sally,099112233,A"
s.split(',').each { |x| puts x }
```

- s.split(x) splits the string according to delimiter x, invoking the code block on each segment

("delimiter" = symbol used to denote boundaries)

Code Blocks for Files

```
File.open("test.txt", "r") do |f|
  f.readlines.each { |line| puts line }
end
```

recall alternative syntax: do … end instead of { … }

- open method takes code block with file argument
  - File automatically closed after block executed
- readlines reads all lines from a file and returns an array of the lines read
  - Use each to iterate
- Can do something similar on strings directly:
- "r1\nr2\n\nr4".each_line { |rec| puts rec }
  - Apply code block to each newline-separated substring

Standard Library: File

- Lots of convenient methods for IO

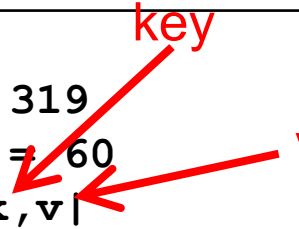
```
File.new("file.txt", "rw")    # open for rw access
f.readline                    # reads the next line from a file
f.readlines                   # returns an array of all file lines
f.eof                         # return true if at end of file
f.close                       # close file
f << object                   # convert object to string and write to f
$stdin, $stdout, $stderr      # global variables for standard UNIX IO
```

  By default, $stdin reads from keyboard, and $stdout and $stderr both write to terminal

- File inherits some of these methods from IO

Code Blocks for Hashes

```
p = {}
p["USA"] = 319
p["Italy"] = 60
p.each { |k,v|
  puts "pop. of #{k} is #{v} million"
}
```




```
pop. of USA is 319 million
pop. of Italy is 60 million
```

- Can iterate over keys and values separately

```
p.keys.each { |k|
    print "key: ", k, " value: ", p[k]
  }
p.values.each { |v|
    print "value: ", v
  }
```

Using Yield to Call Code Blocks

- Any method call can include a code block
  - Inside the method, the block is called with yield
- After the code block completes
  - Control returns to the caller after the yield instruction

```
def twocalls
  return "No block" unless block_given?
  yield
  yield
end
twocalls
twocalls { puts "foo" }
```

```
No block
foo
foo
```

Yield Can Take an Argument

```
def countx(x)
  for i in (1..x)
     puts "foo"
     yield i
  end
end

countx(4) { |x| puts x }
```

```
foo
1
foo
2
foo
3
foo
4
```

- `yield` can take any number of arguments
  - Code block `{|x,y| …}` invoked via `yield` *arg1,arg2*
  - Code block `{|x,y,z| … }` would be invoked via `yield` *arg1,arg2,arg3*
  - Etc.

Quiz 3: What is the output

```
def myFun(x)
  yield x
end
myFun(3) { |v| puts "#{v} #{v*v}" }
```

A. 3
B. 3 9
C. 9 81
D. 9 nil

Quiz 3: What is the output

```
def myFun(x)
  yield x
end
myFun(3) { |v| puts "#{v} #{v*v}" }
```

A. 3

B. **3 9**

C. 9 81

D. 9 nil

Code Blocks are *not* Objects

- Code blocks are limited in their use
  - They cannot be stored in variables, or passed to or returned from methods

```
a = [1,2,3]
a.collect! { |z| z+1 }  # ok
y = { |z| z+1 }  # syntax error
a.collect! y     # syntax error
```

- Only code block literals are permitted, and can only be passed as the last "argument"
  - And only *one* code block, not more

Procs: First-class "code blocks"

- **`Proc`** can make an object out of a code block
  - `t = Proc.new {|x| x+2}`
- **`Proc`** objects can be passed around, stored, and have their code invoked via **`call`**

```
def say(p)
  p.call 10
end

puts say(t)
```

```
12
```

Procs are a Little Clumsy

- Stringing them together is a little (syntactically) heavyweight
  - We will see with OCaml a better integration into the language

```
def say(y)
  t = Proc.new {|x| Proc.new {|z|
                    puts("x=#{x},y=#{y},z=#{z}")
                    z+x+y }}
  return t
end
s = say(2).call(3)
puts s.call(4)
```

```
x=3,y=2,z=4
9
```

Procs vs. code blocks

Code block

- Lightweight syntax
- Common in libraries, programming idioms
- "Second class" status
  - Can only be last, implicit function argument, as a literal
  - Can invoke only from within called method
    - Can't make one and call it in the same method

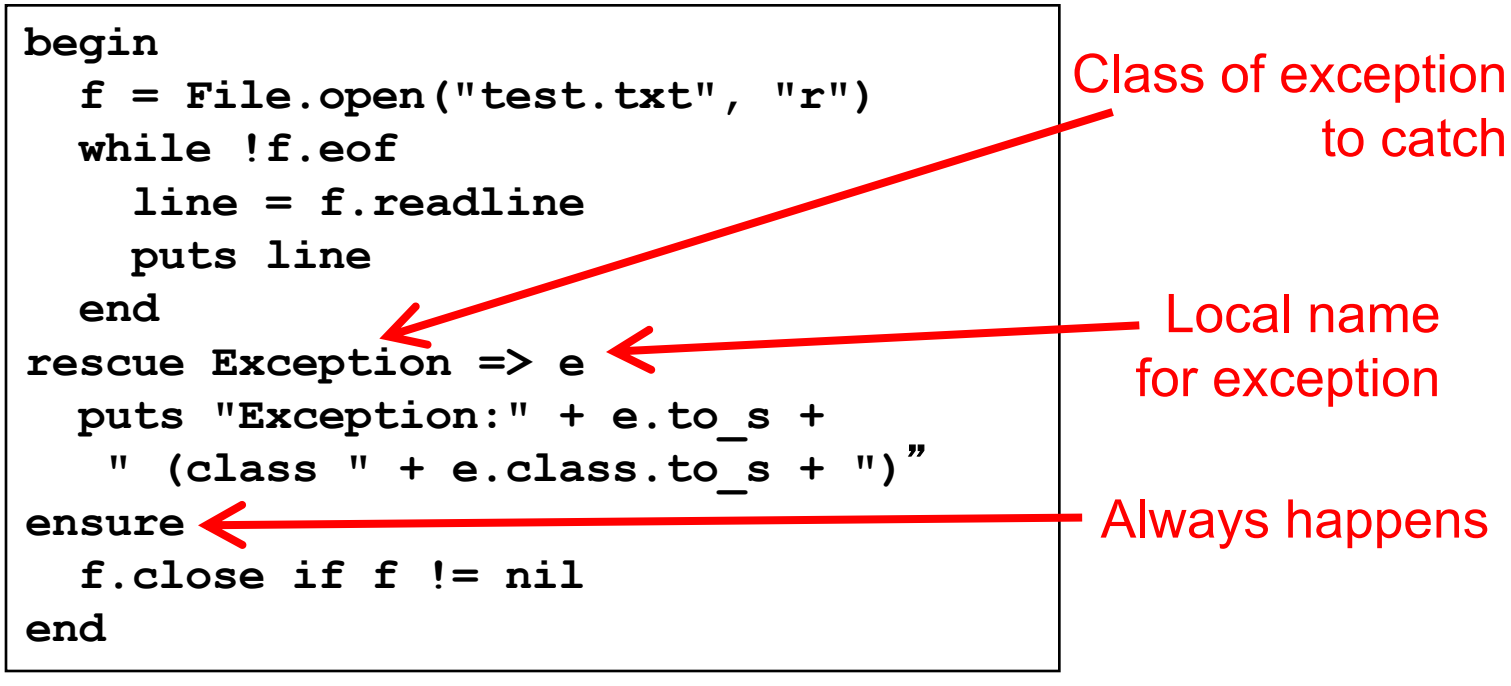
Proc

- Heavier-weight syntax: Must make a Proc from code block first
- Not commonly used in standard libraries
- "First class" status
  - Can pass as argument (or more than one), return as result, store in fields, etc.
  - Call anywhere, directly

Exceptions

- Use begin...rescue...ensure...end
  - Like try...catch...finally in Java

```
begin
  f = File.open("test.txt", "r")
  while !f.eof
    line = f.readline
    puts line
  end
rescue Exception => e
  puts "Exception:" + e.to_s +
   " (class " + e.class.to_s + ")"
ensure
  f.close if f != nil
end
```



Command Line Arguments

- Stored in predefined global constant ARGV

- Example
  - If
    - Invoke test.rb as “ruby test.rb a b c”
  - Then
    - ARGV[0] = “a”
    - ARGV[1] = “b”
    - ARGV[2] = “c”