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

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

code block delimited by `{ }`'s or do...end

parameter name (optional)

body
So, What Are Code Blocks?

- A code block is 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 methods as arguments
Quiz 1: What is the output

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

A. 10
B. 100
C. (Nothing)
D. Error
Quiz 1: What is the output

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

A. 10 – the array itself is not modified by each
B. 100
C. (Nothing)
D. Error
More Code Blocks for Arrays

- Sum up the elements of an array with `each`

```ruby
a = [1,2,3,4,5]
sum = 0
a.each { |x| sum = sum + x }
printf("sum is %d\n", sum)
```

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

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

A. 10
B. 100
C. (Nothing)
D. Error
Quiz 2: What is the output

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

A. 10
B. 100
C. (Nothing)
D. Error
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

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

- Can iterate over keys and values separately
  ```ruby
  population = {}
population["USA"] = 319
  population["Italy"] = 60
  population.each { |c,p|
    puts "population of #{c} is #{p} million"
  }
  ```
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

```ruby
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, at that (not 2, 3, …)
- What about calling them from your methods?
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

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

countx(4) { puts "foo" }
```

```plaintext
1
foo
2
foo
3
foo
4
foo
```
Yield Can Take an Argument

- It can take any number of arguments
  - Code block \{ |x,y| \ldots \} invoked via \texttt{yield arg1,arg2}
  - Code block \{ |x,y,z| \ldots \} would be invoked via \texttt{yield arg1,arg2,arg3}
  - Etc.

```ruby
def do_it_twice
  return "No block" unless block_given?
  yield "hello"
  yield "there"
end

do_it_twice { |x| puts x }
```

hello
there
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
**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`

```ruby
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

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

```ruby
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”