CS 360 Internet Programming

Ruby

More Ruby

Daniel Zappala
Computer Science
Brigham Young University

Handling Exceptions

```
opfile_name = "/tmp/testfile"
socket = $stdin
op_file = File.open(opfile_name, "w")
begin
  while data = socket.read(512)
    op_file.write(data)
  end
rescue SystemCallError
  $stderr.print "IO failed: " + $!
  op_file.close
  File.delete(opfile_name)
  raise
end
```

- exceptions in the block are caught by rescue clause
- multiple resucue clauses allowed, handled like a case statement
- `raise` reraises the exception

Ensure Clause

- runs a block of code regardless of exceptions

```
f = File.open("testfile")
begin
  # .. process
rescue
  # .. handle error
ensure
  f.close unless f.nil?
end
```

Else Clause

- runs a block of code if no exceptions raised

```
f = File.open("testfile")
begin
  # .. process
rescue
  # .. handle error
else
  puts "Congratulations— no errors!"
ensure
  f.close unless f.nil?
end
```

Retry Statement

```
@esmtp = true
begin
  # First try an extended login. If it fails because the
  # server doesn't support it, fall back to a normal login
  if @esmtp then
    @command.ehlo(helodom)
  else
    @command.helo(helodom)
  end
rescue ProtocolError
  if @esmtp then
    @esmtp = false
    retry
  else
    raise
  end
end
```

Raising Exceptions

```
raise
raise "bad mp3 encoding"

class InterfaceException < RuntimeError
end
raise InterfaceException, "Keyboard failure", caller
```

1. RunTimeError: default Exception class
2. passes a message to the rescue clause
3. raises a user-specified Exception, with a message and a stack trace

Catch and Throw

```
songlist = ""
def songlist.play() end
catch (:done)  do
  while line = gets
    throw :done unless fields = line.split(/\t/)
    songlist.add(Song.new(*fields))
  end
  songlist.play
end
```

- `throw` passes control back to the catch block

Defining Modules

- a module creates a namespace for a set of methods and classes

```
module Gnuplot
  def Gnuplot.open( persist=true )
    cmd = Gnuplot.gnuplot( persist ) or raise 'gnuplot not found'
    IO::popen( cmd, "w") { |io| yield io }
  end
  class Plot
    attr_accessor :cmd, :data, :sets
    def initialize (io = nil, cmd = "plot")
    ...
  end
end
```

Using Modules

- call the methods just like a class method
- use a class by prefixing with the module name

```
require 'gnuplot'
Gnuplot.open do |gp|
  Gnuplot::Plot.new( gp ) do |plot|

    plot.xlabel "Load (Sessions/s)"
    ...
```

Mixins

```
module Debug
  def who_am_i?
    "#{self.class.name} (\##{self.id}): #{self.to_s}"
  end
end
class Phonograph
  include Debug
  # ...
end
ph = Phonograph.new("West End Blues")
ph.who_am_i? -> "Phonograph (#945760): West End Blues"
```

- all instance methods defined in the module become available as class methods
- replaces mutliple inheritance
- must use require first if method is not in the same file

Helpful Mixins

- Comparable module
  - define the <=> method in your class (returns -1 if less than, 0 if equal, 1 if greater than)
  - include Comparable
  - get <, <=, ==, >=, >, `between?` from the module
- Enumerable module
  - define an iterator called each that returns the elements of your collection
  - include Enumerable
  - get the map, include?, find_all?, inject iterators
  - define <=> and also get min, max, and sort

Ruby Threads

```ruby
require 'net/http'
pages = %w( www.rubycentral.com slashdot.org www.google.com )
threads = []
for page in pages
  threads << Thread.new(page) do |url|
    h = Net::HTTP.new(url, 80)
    puts "Fetching: #{url}"
    resp = h.get('/', nil )
    puts "Got #{url}:  #{resp.message}"
  end
end
threads.each {|thr|  thr.join }
```

- user-level threads
- use block parameter to create a local variable, because page will get overwritten each time through the loop

Thread Variables

- threads can store per-thread state accesible by other threads

```
srand 2
count = 0
threads = []
10.times do |i|
  threads[i] = Thread.new do
    sleep(rand(0.1))
    Thread.current["mycount"] = count
    count += 1
  end
end
threads.each {|t| t.join; print t["mycount"], ", " }
puts "count = #{count}"
```

Unsafe Code

```
class Counter
  attr_reader :count
  def initialize
    @count = 0
    super
  end
  def tick
    @count += 1
  end
end
c = Counter.new
t1 = Thread.new { 10000.times {  c.tick } }
t2 = Thread.new { 10000.times {  c.tick } }
t1.join
t2.join

puts c.count -> 14268
```

Using a Monitor

```
require 'monitor'
class Counter < Monitor
  # ...
  def tick
    synchronize do
      @count += 1
    end
  end
end
c = Counter.new
t1 = Thread.new { 10000.times {  c.tick } }
t2 = Thread.new { 10000.times {  c.tick } }
t1.join
t2.join

puts c.count -> 200000
```

Thread-Safe Queue

- the thread module defines a thread-safe queue class

```
require 'thread'
queue = Queue.new
Thread.new do
  obj = queue.deq
  # ...
end
Thread.new do
  obj = ...
  queue.enq(obj)
end
```

Condition Variables

- use signal to indicate the condition has occurred

```
playlist = []
playlist.extend(MonitorMixin)
pending = playlist.new_cond
customer = Thread.new do
  loop do
    req = customer_request
    playlist.synchronize do
      playlist << req
      pending.signal
    end
  end
end
```

Condition Variables

- use wait_while to wait while a condition is not true

```
player = Thread.new do
  loop do
   playlist.synchronize do
     pending.wait_while { playlist.empty? }
     song = playlist.shift
   end
   play(song)
end
```

Spawning Processes

- use system or backticks

```
system("tar xzf test.tgz")
result = `date`
```

Using Pipes

```
pig = IO.popen("/usr/local/bin/pig","w+")
pig.puts "ice cream after they go to bed"
pig.close_write
puts pig.gets

-> iceway eamcray afterway eythay ogay otay edbay
```

- `close_write` forces the pipe to flush the output

Using Exec

```
exec(web) if fork.nil?
...
system("kill `cat web.pid`")
```