CS 360 Internet Programming

Ruby

More Ruby

Daniel Zappala
Computer Science
Brigham Young University
1. Exceptions
   - Handling Exceptions
   - Raising Exceptions
   - Catch and Throw

2. Modules
   - Defining Modules
   - Using Modules
   - Mixins

3. Threads and Processes
   - Threads
   - Mutual Exclusion
   - Processes
Handling Exceptions

```ruby
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 rescue clauses allowed, handled like a case statement
- `raise` reraises the exception
Ensure Clause

- runs a block of code regardless of exceptions

```ruby
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

```
1  f = File.open("testfile")
2  begin
3    # .. process
4    rescue
5      # .. handle error
6  else
7      puts "Congratulations— no errors!"
8  ensure
9      f.close unless f.nil?
10  end
```
1  @esmtp = true
2  begin
3    # First try an extended login. If it fails because the
4    # server doesn’t support it, fall back to a normal login
5      if @esmtp then
6        @command.ehlo(helodom)
7      else
8        @command.helo(helodom)
9      end
10     rescue ProtocolError
11      if @esmtp then
12        @esmtp = false
13        retry
14      else
15        raise
16      end
17     end
18  end
Raising Exceptions

1. `raise`
2. `raise "bad mp3 encoding"
3. `class InterfaceException < RuntimeError
4. `end`
5. `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

```ruby
songlist = ""
def songlist.play() end

begin
  catch (:done) do
    while line = gets
      throw :done unless fields = line.split(/\s+/)
      songlist.add(Song.new(*fields))
    end
  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

```ruby
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
end
```
Using Modules

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

```ruby
1  require 'gnuplot'
2  Gnuplot.open do |gp|
3    Gnuplot::Plot.new(gp) do |plot|
4      plot.xlabel "Load (Sessions/s)"
5  ...  
```
Mixins

```ruby
1 module Debug
2   def who_am_i?
3     "#{self.class.name} (#{self.id}): #{self.to_s}"
4   end
5 end
6 class Phonograph
7   include Debug
8   # ...
9 end
10 ph = Phonograph.new("West End Blues")
11 ph.who_am_i? => "Phonograph (#945760): West End Blues"
```

- all instance methods defined in the module become available as class methods
- replaces multiple 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 accessible by other threads

```ruby
1 srand 2
2 count = 0
3 threads = []
4 10.times do |i|
5   threads[i] = Thread.new do
6     sleep(rand(0.1))
7     Thread.current["mycount"] = count
8     count += 1
9   end
10 end
11 threads.each { |t| t.join; print t["mycount"], ", " }
12 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

```ruby
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
```
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

```ruby
1 playlist = []
2 playlist.extend(MonitorMixin)
3 pending = playlist.new_cond
4 customer = Thread.new do
5   loop do
6     req = customer_request
7     playlist << req
8     pending.signal
9   end
10  end
11 end
12 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
```

Daniel Zappala
Computer Science
Brigham Young University
CS 360 Internet Programming
Spawning Processes

- use system or backticks

```c
1  system("tar xzf test.tgz")
2  result = 'date'
```
Using Pipes

```python
1  pig = IO.popen("/usr/local/bin/pig" ,"w+")
2  pig.puts "ice cream after they go to bed"
3  pig.close_write
4  puts pig.gets
5
6  --> icecream after they go to bed
```

- `close_write` forces the pipe to flush the output
Using Exec

```c
1   exec(web) if fork.nil?
2   ...
3   system("kill 'cat web.pid'")
```