Sesi 3: Advanced Topic

Exceptions

Exception terjadi ketika aliran program terinterupsi karena error. Penanganan exception pada Ruby mirip dengan C++ dan Java.

C++	Java	Ruby
try {}	try{}	begin/end
catch {}	catch {}	rescue
	finally	ensure
throw	throw	raise

Ketika terjadi error, baris selanjutnya tidak akan dieksekusi. Ruby akan menangkap error tersebut dengan `rescue`, dan akhirnya menjalankan instruksi `ensure`.

```
def foo
  str = "Hello world"
  str.bar
  puts "This line is ignored"
rescue NoMethodError => e
  print "Catching: " + e.to_s
rescue Exception
  print "Problem found!"
ensure
  puts " -- This is it"
end

foo # => Catching: undefined method `bar' for "Hello world":String -- This is it
```

Kita bisa melemparkan exception kapan saja dengan `raise`.

```
def goo(n)
  if n < 0
    raise StandardError, "Negative number"
  else
    puts "It's positive"
  end
end

def hoo
  goo(-1)
  puts "It is printed?"
rescue Exception => e
  print "Exception: " + e.to_s
end

hoo # => Exception: Negative number
```

Selain menggunakan keyword `rescue` dan `raise`, kita bisa juga menggunakan method `catch` dan `throw` pada Ruby. Penggunaan argumen `catch` dan `throw` pada Ruby ini berbeda dengan di

Java, meskipun namanya sama.

```
def print_me(n)
  puts n
  throw :done if n <= 0
  print_me(n-1)
end

catch(:done) do
  print_me(5)
end
```

Penggunaan `throw` dan `catch` di ruby sangat jarang dilakukan dan sebaiknya dihindari

Advanced Blocks

Implicit Block

Setiap method di ruby dapat menerima block sebagai argumen terakhir, sekalipun dalam definisi method tersebut tidak mencantumkan argumen block.

Untuk mengecek apakah terdapat sebuah argumen block, panggil `block_given?` dalam sebuah method. Sedangkan untuk mengeksekusi block tersebut dapat menggunakan `yield`.

```
def foo
  if block_given?
    yield
    yield
  else
    puts 'No block here'
  end
end

foo { print 'Hello ' } # => Hello Hello

foo # => No block here
```

`yield` dapat menerima argumen yang akan diteruskan ke block yang dieksekusi

```
def foo
  yield([1,2,3])
end

foo do |array|
  puts array
end
```

Proc

Ruby memungkinkan kita menyimpan *procedure* atau `Proc` sebagai objek lengkap dengan konteksnya.

Penggunaan *block* pada Ruby itu relatif sederhana. Meskipun demikian, kita mungkin perlu punya beberapa *block* yang berbeda dan menggunakannya berkali-kali. Untuk menghindari kode yang repetitif tersebut, kita bisa menyimpannya sebagai *procedure* atau `Proc`, sebagai objek yang

lengkap dengan konteksnya. Perbedaan antara *block* dan `Proc` adalah *block* merupakan suatu `Proc` yang tidak bisa disimpan dan merupakan solusi sekali pakai.

```
def foo
  label = proc do
    print "counting"
  end

  count = Proc.new do
    [1,2,3,4,5].each do |i|
      print i
    end
    puts
  end

  label.call
  print " - "
  count.call
end

foo # => counting - 12345
```

Lambda

Di Ruby, *lambda* merupakan suatu *method* yang menciptakan objek *Proc* yang terkait dengan konteks saat ini dan melakukan pengecekan parameter ketika dipanggil.

Berbeda dengan Proc, lambda akan mengecek jumlah argumen yang dioper.

```
def foo(code)
  one, two = 1, 2
  code.call(one, two)
end

foo(Proc.new { |a, b, c| puts "a = #{a}, b = #{b}, c = #{c.class}" } ) # => a =
1, b = 2, c = NilClass

foo(lambda { |a, b, c| puts "a = #{a}, b = #{b}, c = #{c.class}" } ) # =>
ArgumentError: wrong number of arguments (2 for 3)
```

Passing Proc Around

Untuk mem-pass suatu proc atau lambda sebagai argumen suatu method, variabel proc/lambda tersebut diberi prefix &

```
print_item = proc do |item|
  puts item
end

[1,2,3].each &print_item
```

Classes as Objects

Object merupakan *root* dari hierarki kelas di Ruby. Semua *method* dari kelas *Object* akan tersedia di seluruh kelas kecuali di-*override* secara eksplisit. Selengkapnya ada di http://www.ruby-

doc.org/core/classes/Object.html

Metaprogramming

Metaprogramming merupakan pemrograman yang memanipulasi program lainnya atau bahkan dirinya sendiri, atau mengerjakan hal-hal yang biasanya dilakukan pada *compile time*, pada waktu *runtime*. Kemampuan suatu objek untuk memanipulasi dirinya sendiri ini sering disebut sebagai *reflection*.

Metaprogramming di ruby memanfaatkan sifat ruby yang sangat dinamis. Beberapa teknik yang sering digunakan dalam metaprogramming di ruby antara lain:

Monkey Patching

Salah satu bentuk *metaprogramming* adalah *monkey patching*, yaitu memodifikasi *runtime code* dari suatu bahasa yang dinamis tanpa mengubah *source code* aslinya.

Contoh yang sangat menarik dalam monkey patching di ruby adalah dengan melakukan overwrite kelas yang sudah didefinisikan sebelumnya, termasuk kelas yang built-in (bawaan) ruby seperti `Fixnum`. Kelas dalam ruby dapat dibuka dan didefinisikan ulang kapanpun kita mau. Dengan demikian kita dapat dengan mudah mengubah behaviour dari suatu object sesuai keinginan kita.

Dibawah ini adalah contoh monkey patching yang sangat berbahaya. Di sini kita membuka kembali kelas `Fixnum` yang merupakan kelas bagi angka integer bawaan ruby. Kemudian kita mendefinisikan ulang method `+` untuk selalu me-return angka 42 kapanpun.

```
class Fixnum
  def +(x)
    42
  end
end

3 + 4 # => 42
```

Kita bahkan juga bisa meng-*override* semua objek dengan cara meng-override kelas `Object` yang merupakan hirarki kelas tertinggi, sehingga semua objek akan terpengaruh.

Contoh berikut ini menambahkan atribut timestamp bagi semua object, sehingga setiap object akan menyimpan waktu ketika object tersebut diinisiasi.

```
class Object
  attr_accessor :timestamp
end

class Class
  alias_method :old_new, :new
  def new(*args)
    result = old_new(*args)
    result.timestamp = Time.now
    result
  end
end
```

```
class Foo
end

Foo.new.timestamp
```

method_missing

Ketika kita mengirimkan pesan ke objek, objek akan mengeksekusi *method* pertama yang ditemukannya pada *method lookup path* dengan nama yang sama persis dengan pesan. Jika tidak ditemukan, ia akan melemparkan *exception* NoMethodError, kecuali kalau kita sudah menyediakan sebuah *method* bernama `method_missing`.

`method_nissing` dapat kita override sesuai dengan keinginan kita. Misalnya kita dapat meng-override `method_missing` sehingga jika kita memanggil method yang tidak tersedia, object tersebut akan memberi kita peringatan.

```
class Foo
  def method_missing(m, *args, &block)
    puts "method #{m} is not defined. Please define it first."
  end
end

Foo.new.bar # => method bar is not defined. Please define it first.
```

I/O

Dasar dari semua input/output di Ruby adalah kelas `IO`, di mana merepresentasikan stream input/output data dalam bentuk byte. Standard stream mencakup stdin (keyboard), stdout (display ke layar) dan stderr (error output). Parameter pertama konstruktor kelas IO adalah nomer *file descriptor*, sedangkan parameter keduanya merupakan mode I/O. Ada beberapa macam mode I/O, antara lain: r, r+, w, w+, a, a+, b

Stream	File Descriptor
stdin	0
stdout	1
stderr	2

```
ios = IO.new( 1, "w" )
ios.puts "Hello IO" # => Hello IO

ios.fileno # => 1
ios.to_i # => 1

ios.close
```

Files

Di Ruby, kita bisa memanipulasi file dan folder dengan menggunakan *method-method* dari kelas Dir dan File.

Buat dan hapus folder:

```
Dir.mkdir("/tmp/ruby_basic",777)
dir = Dir.pwd
puts dir
Dir.rmdir("/tmp/ruby_basic")
```

Buat dan hapus file:

```
file = File.new( "aaaa.txt", "w" )
File.rename( "aaaa.txt", "bbbb.txt" )
File.delete( "bbbb.txt" )
```

Pemrosesan file per baris pada Ruby:

```
file = File.open( "Readme.md" ) if File::exists?( "Readme.md" )

file.each do |line|
  print "#{file.lineno}. ", line
end unless file.closed?

file.close
```