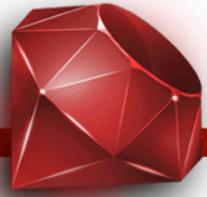




Introducción a Ruby

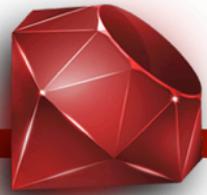
Rodrigo Rodriguez
rorodr@gmail.com
Skype: rarodriguezr

Manejo de Fechas



- Existen 3 clases distintas para el manejo de Fechas: Date, Time y DateTime
- Por defecto las clases Time y DateTime utilizan la hora/zona horaria del computador donde se obtiene la información
- Todas las clases cuentan con un método “parse”, el cual convierte un texto en un elemento de tipo fecha.

Manejo de Fechas



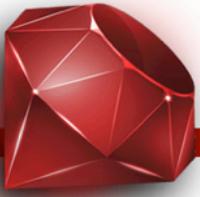
```
t = Time.now  
# obtener información de una sección del tiempo  
puts t.sec  
puts t.wday  
puts t.utc  
  
# Algunos cálculos  
puts t + 10.hours  
puts t - 5.minutes  
  
# dar formato al texto  
puts t.strftime("%A")  
puts t.strftime("%B")  
# %A - día de la semana  
# %B - mes del año  
puts t.strftime("son las %H:%M %Z")  
puts t.strftime("%d/%m/%Y %H:%M:%S")
```

Manejo de Fechas



- Algunos métodos importantes:
 - **date1 – date2 (ó +)**: Total de segundos entre una fecha y otra
 - **strptime**: Semejante al parse pero uno le puede establecer el formato de la fecha
 - `DateTime.strptime("{ 2009, 4, 15 }", "{ %Y, %m, %d }")`
 - **strftime**: Permite dar formato a la fecha por presentar en pantalla

Trabajo 0: Manejo de Fechas



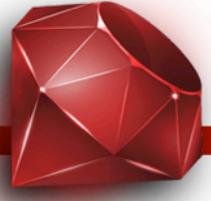
- Crear un método que reciba una fecha con formato: “31/05/2015 19:20:00”, convertir el texto a fecha y hacer la siguiente verificación:
 - Si los minutos, segundos u horas son múltiplos de 10, imprimir: “Una hora especial: HORA”
 - Si los días o meses son múltiplos de 7 imprimir: “Una fecha especial: FECHA”
 - Si la fecha es mayor a hoy: imprimir la fecha en formato: “Hoy es DIA de MES del AÑO y son las HORA PM”

Símbolos



- Usados para identificar elementos o llaves
- Cada símbolo tiene un nombre e ID único
- Símbolos no pueden contener información solamente son etiquetas
- Todos los símbolos están almacenados en una “tabla de símbolos”
 - :symbol_example.object_id # 132178
 - :symbol_example.object_id # 132178

Hashes (1/3)



- Son arreglos asociativos que pueden usar cualquier valor como llave
- Por lo general las llaves de un arreglo son símbolos o strings
- Los elementos de un Hash no están ordenados y dependen del orden en que se agregan

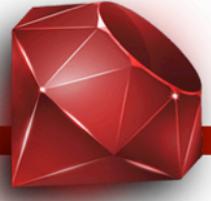
```
person = {}
person[:name] = 'John Connor'
person.store :age, 26
puts person.keys.inspect
puts person.values.inspect
puts person[:name]
```

Hashes (2/3)



- Los hashes son de las estructuras de datos más utilizadas en Rails
- En Rails veremos que hay 2 tipos adicionales de Hashes: Ordenados y con acceso indiferente
 - Hash ordenado: se ordena a partir de la llave
 - Hash de acceso indiferente: Permite accesar a sus valores tanto por la llave en string o en símbolo.

Hashes (3/3)



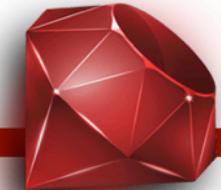
- Por lo general, la generación de JSON se facilita mucho desde un hash
- Algunos métodos comunes son:
 - each_key, each_value, each_pair, each: iteradores
 - values, keys: Obtener las llaves y los valores
 - merge, merge!: mezclar 2 hashes

Trabajo 1: Hashes



- Considerando el siguiente arreglo:
 - users = [{name: “Juan”, last_name: “Perez”, children: [“Ana”, “Pablo”]}, {name: “Martina”, last_name: “Juarez”, children: nil}, ...]
- Imprimir en pantalla los detalles de cada uno de los usuarios. En caso de que no tengan hijos, se deberá de mostrar el mensaje: “Sin hijos”.

Clases (1/3)



- Todas las clases en Ruby heredan de Object
- Definir una clase en Ruby es simple, solamente es necesario agregar “class” seguido del nombre de la clase en formato CamelCase.
- El método inicializador de la clase es llamado “initialize”
- Para crear una nueva instancia de una clase se utiliza:
`Class.new(PARAMS)`

Clases (2/3)



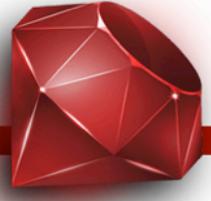
```
class Dog
attr_accessor :breed
def initialize(breed, name)
  @breed = breed
  @name = name
end
def bark
  'guau! guau!'
end
def greed
  "Soy un #{@breed} y mi nombre es #{@name}"
end
d = Dog.new('Collie', 'Lassie')
puts d.greed
puts d.bark
```

Clases (3/3)



- Se pueden declarar métodos de clase utilizando la palabra ‘self’ antes de la declaración del método. Ej:
 - “def self.list_all”
- Algunos métodos interesantes para todo objeto:
 - respond_to?: verifica si el método tiene un método específico. Ej: string.respond_to?("patito")
 - instance_of? is_a? Verifica si la instancia es de una clase determinada. Ej: num = 10; num.is_a? Fixnum
 - object_id: Permite ver el identificador el objeto

Clases: Accessors



- Permiten accesar a las variables de la instancia desde fuera del objeto
- Se puede permitir la lectura, escritura o ambas

```
# accesor de lectura  
attr_reader :title, :artist
```

```
# accesor de escritura  
attr_writer :title
```

```
# accessor de lectura y escritura  
attr_accessor :name
```

Clases: Control de Acceso



- Ruby permite 3 niveles de acceso en las clases:
 - **Public:** cualquiera puede accesar esos métodos.
Este es el valor por defecto en las clases nuevas.
 - **Protected:** Estos métodos solo puede ser accesados por instancias de la clase y sus subclases.
 - **Private:** Estos métodos solo pueden ser utilizados por el mismo objeto (`self`)

Clases: Herencia (1/2)



- Las clases heredan los métodos y características que tienen sus padres
- La herencia es declarada con el uso de “<” en la declaración de la clase. Ej.:
 - Class Cat < Animal
- Es posible sobreescribir todos los métodos de la clase padre
- El método “super” busca por un método con el mismo nombre y la misma cantidad de parámetros en la clase padre.

Clases: Herencia (2/2)



```
class Bicicleta
attr_reader :marchas, :ruedas, :asientos
def initialize(marchas = 1)
  @ruedas = 2
  @asientos = 1
  @marchas = marchas
end
end

class Tandem < Bicicleta
def initialize(marchas)
  super
  @asientos = 2
end
end
```

t = Tandem.new(2)
puts t.marchas
puts t.ruedas
puts t.asientos
b = Bicicleta.new
puts b.marchas
puts b.ruedas
puts b.asientos

Clases: Modificar clase

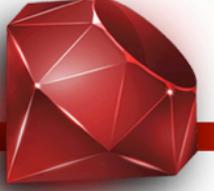


- En Ruby, las clases nunca están cerradas: siempre se pueden añadir métodos.
- Nada más hay q continuar con la declaración de la clase.

```
class String
  def num_caracteres
    puts self.size
  end
end
```

```
texto = 'Cielo empedrado, suelo mojado'
texto.num_caracteres
```

Clases: Sobrecarga de métodos



- En Ruby la sobrecarga no es factible de hacer pues sólo se puede tener un método con un nombre dado

```
def test s  
  puts "hola test #{s}"  
end
```

```
def test  
  puts "Nueva version de test"  
end
```

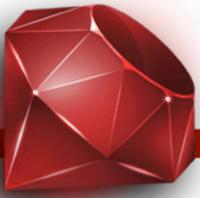
```
test          # Se imprime "goodbye test"  
test "my string" # Causa un error:  
                  # wrong # of arguments (1 for 0) (ArgumentError)
```

Clases: Sobrecarga de métodos



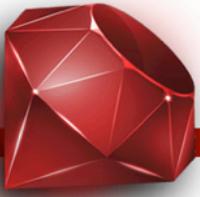
```
class Rectangle
  def initialize(*args)
    if args.size < 2 || args.size > 3
      # se puede mostrar una excepcion aca
      puts 'This method takes either 2 or 3 arguments'
    else
      if args.size == 2
        puts 'Two arguments'
      else
        puts 'Three arguments'
      end
    end
  end
end
Rectangle.new([10, 23], 4, 10)
Rectangle.new([10, 23], [14, 13])
```

Clases: Congelar un objeto



- En Ruby se puede “congelar” un objeto, de modo que éste no pueda ser modificado posteriormente.
- Para ello se usa “freeze”, lo que lo deja en el estado inmutable, siendo la única forma de quitar ese estado re-crear el objeto o duplicarlo (puede ser en el mismo nombre de variable).
- Si se intenta modificar el objeto, se obtiene una excepción

Clases: Duplicar un objeto



- En Ruby se pueden duplicar objetos utilizando 2 métodos:
 - clone**: literalmente clona el objeto, incluyendo los estados que tenga el objeto (por ejemplo el `freeze`), así como métodos agregados específicamente a la instancia del objeto.
 - dup**: crea un objeto nuevo y comparte los valores de cada uno de los atributos existentes. Es decir, si se modifica el “duplicado”, se va ver afectado el original.

Clases: Duck typing



- Duck Typing se refiere a la tendencia de Ruby a centrarse menos en la clase de un objeto, y dar prioridad a su comportamiento: qué métodos se pueden usar, y qué operaciones se pueden hacer con él.
- Se utiliza el método “respond_to?” para verificar si la clase tiene ese metodo o no:
 - “mi string”.respond_to?(:to str)

Clases: Duck typing

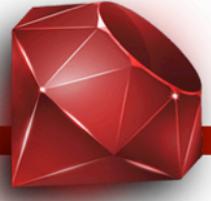


```
class Duck
  def quack
    'Quack!'
  end
  def swim
    'Paddle paddle paddle...'
  end
end
```

```
class Goose
  def honk
    'Honk!'
  end
  def swim
    'Splash splash splash...'
  end
end
```

```
def make_it_quack(duck)
  duck.quack if duck.respond_to?(:quack)
end
puts make_it_quack(Duck.new)
puts make_it_quack(Goose.new)
```

Trabajo 2: Clases



1. Crear una clase llamada MyCar. Cuando se inicializa una nueva instancia de la clase, permitir al usuario definir algunas variables de instancia, para poder acceder al año, color y modelo del carro. Además establecer en 0 la velocidad actual. Por otro lado, tienen que existir métodos de instancia para acelerar, frenar y apagar el carro

Trabajo 2: Clases



2. Agregar a la clase MyCar, un método de clase que permita calcular el gasto de gasolina por kilómetro de cualquier carro (recibe 2 parámetros: kilometraje realizado y litros de gasolina gastados)
3. Crear una super clase llamada “Vehicle”, de la cual hereda MyCar. Se debe de mover a esta clase los comportamientos no específicos de MyCar. Además se debe de crear una Constante que diferencie a MyCar de otros tipos de vehículos y un método que permita imprimirla (Vehicle). Además, crear la clase MyTruck que hereda de vehículo y tiene su propia constante que la diferencia de otros vehículos.

Trabajo 2: Clases



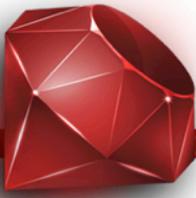
4. Agregar a la superclase alguna manera para llevar control de la cantidad de vehículos creados (sin importar la subclase que corresponda) y a la vez crear un método para imprimir dicho valor en pantalla.

Expresiones regulares



- Permite reconocer patrones en un texto
- Si se desea encontrar un patrón en un texto, se haría:
 - `m1 = "Ruby: a powerful language".match /Ruby/`
 - M1 va a tener un elemento de tipo “MatchData” con el texto que coincide con lo buscado
 - `m2 = "El futuro es Ruby" =~ /Ruby/`
 - #m2 contains the position of the expression

Expresiones Regulares



Expression	Meaning
.	Cualquier caracter
[]	Especificar rangos
\w	Letra o número
\W	Cualquier caracter que no sean letras o números
\s	Un caracter de espacio [\t\n\r\f]
\S	Cualquier caracter que no sea un espacio
\d	Número
*, ?	Cero o más repeticiones
+	Una o más repeticiones
\$ - \z	Fin de línea
{m,n}	Al menos M elementos y máximo N
()	Agrupar Elementos
	Operador lógico OR

Trabajo 3: Expresiones regulares



1. Asumiendo que tenemos un arreglo con textos semejantes a este, separar cada uno de los elementos (cédula, nombre y fecha de nacimiento)
 - 0-0000-0000Juan Perez Osorio27/06/1973
2. Considerando el siguiente HTML, mostrar en pantalla únicamente el texto usando REGEX
 - <div>Hello worldname's email</div>

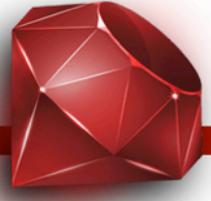
Pista: Pueden usar rubular.com para probar su REGEX

Bloques de código (1/6)



- Un bloque es una sección de código entre “{}” o “do..end”
- Bloques de código pueden recibir parámetros
- Cualquier método puede recibir un bloque de código (por ejemplo, ya vimos que el `delete_if` y `select` reciben un bloque con las condiciones)

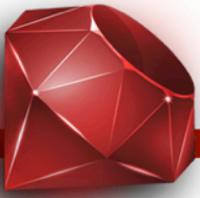
Bloques de código (2/6)



- Un ejemplo que ya vimos fue con el “delete_if” y “select”, a los cuales se les pasa un parámetro con las condiciones de selección
- Otro ejemplo de bloques simples sería:

```
def greet2(texto)
  puts "mi parametro es #{texto}"
  yield
end
greet2 ("argumento") {puts "Hola #{argumento}"}
```

Bloques de código (3/6)



- `yield`:
 - Le dice al método que ejecute el bloque de código que se le provee al método (puede pasar parámetros)
- `block_given?`
 - Verifica si se le provee un bloque a la función o no
- Parámetros:
 - Un bloque puede recibir parámetros de ejecución, para ello se ocupamos enviarlos a la hora de ejecutar el `yield`

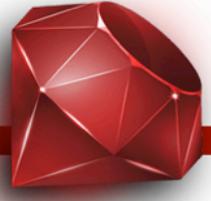
Bloques de código (4/6)



```
def greet2(texto)
  puts "el saludo para mi gente es: #{texto}"
  yield rand(10), rand(50) if block_given?
  puts "un placer haberlo visto"
end
greet2 ("hola!!!!") { |x,y| puts "#{x}, #{y}" }
```

```
def greet2(texto, &my_block)
  puts "el saludo para mi gente es: #{texto}"
  puts my_block.call(rand(10), rand(50))
  puts "un placer haberlo visto"
end
greet2 ("hola!!!!") { |x,y| "#{x}, #{y}" }
```

Bloques de código (5/6)



- Los bloques no son objetos, pero pueden convertirse en ellos gracias a la clase Proc.
- Los objetos tipo proc son bloques que se han unido a un conjunto de variables locales.
- Esto se hace gracias al método lambda del módulo Kernel.

```
prc = lambda{ "hola" }
puts prc.call
```

Bloques de código (6/6)



```
prc = lambda {puts 'Hola'}  
prc.call #llamamos al bloque
```

```
#otro ejemplo  
toast = lambda do  
  puts 'Gracias'  
end  
toast.call
```

Trabajo 4: Bloques

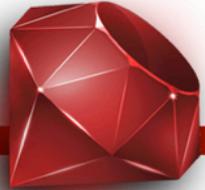


- Hacer un método “suma” que reciba de parámetro un arreglo de números y un bloque
 - Arreglo de prueba: [7,6,5,4,3,2,1]
 - Bloque: {|total, valor| total + valor }
- La clase Array tiene un método llamado “map” o “collect”, utilizar ese método para convertir:
 - [‘agua’, ‘tierra’, ‘aire’, ‘fuego’] en: [{name: ‘agua’, id: 1},{name: ‘tierra’, id:2}, ...]

Modulos (1/3)



- Son similares a las clases (con ciertas diferencias), pueden contener métodos, variables, constantes, Clases y otros elementos.
- No es posible crear clases que hereden de los módulos, pero cualquier clase/módulo puede incluir funcionalidad de otros módulos.
- Los módulos se utilizan para añadir funcionalidades generalizables o librerías



Módulos (2/3)

```
module Trig
  PI = 3.1416
  # métodos
  def Trig.sin(x)
    # ...
  end
  def cos(x)
    # ...
  end
end

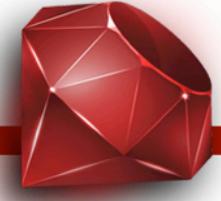
require 'trig'
Trig.sin(Trig::PI/4)
```

```
module D
  def initialize(nombre)
    @nombre = nombre
  end
  def to_s
    @nombre
  end
end
```

```
module Debug
  include D
  def quien_soy?
    "#{self.class.name} (\#\{self.object_id}): #{self.to_s}"
  end
end
```

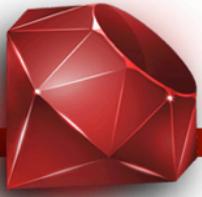
```
class OchoPistas
  include Debug
end
```

Módulos (3/3)



- Existen 2 métodos para incluir funcionalidad de un módulo en una clase:
 - **Include**: Incluye los métodos del módulo como si fueran métodos de instancia.
 - **Extend**: Incluye los métodos del módulo como si fuera métodos de clase (tienen q llamarse usando Clase.mi_metodo)

Usando librerías adicionales



- Para cargar librerías en Ruby se utiliza **require** o **load**
- **Require** carga el archivo únicamente una vez, sin importar si tenemos la misma instrucción más de 2 veces
- **Load:** Lee el archivo cada vez que se utiliza la línea de carga

Trabajo 5: Módulos



- Crear una clase “MiNumero” que incluya los métodos de los módulos “suma” y “resta” (como método de instancia). Además, agregar un módulo “multiplicacion” cuyos métodos serán agregados como métodos de clase.
 - El código generado debe de poder funcionar con esta prueba:
 - mate = MiNumero.new(15)
 - mate.sumar(15) → 30
 - mate.restar(4) → 11
 - MiNumero.multiplicar(4,6) → 24

Manejo de archivos (1/2)



- Manejar archivos y carpetas se realiza por medio de una clase llamada IO, de la cual heredan File, FileUtils y CSV y otras clases.
- Al heredar de IO, comparten muchos de los métodos pues la mayoría se declaran en IO.
- Algunos métodos importantes:
 - **each, each_line, each_char, gets, read**: cargar archivos
 - **putc, puts, print, write**: escribir archivos

Manejo de archivos (2/2)



```
# Abre y lee un fichero. Se usa un bloque: el fichero se cierra
# automáticamente al acabar el bloque.

File.open('fichero.txt', 'r') do |f1|
    while linea = f1.gets
        puts linea
    end
end

# Crea un nuevo fichero, y escribe

File.open('text.txt', 'w') do |f2|
    # '\n' es el cambio de línea
    f2.puts "Por que la vida \n puede ser maravillosa"
end
```

Trabajo 6: Archivos

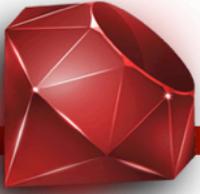


- Descargar el archivo:

[https://github.com/rarodriguez/curso_rails/tree/
master/trabajos%20en%20clase/
sacramentocrime_january2006.csv](https://github.com/rarodriguez/curso_rails/tree/master/trabajos%20en%20clase/sacramentocrime_january2006.csv)

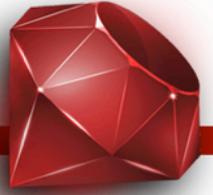
- Crear un método que cargue la información de ese archivo y genere un archivo nuevo que contenga únicamente:
 - Día (ignoramos hora), Distrito, “Total delitos en ese día en ese distrito”

Excepciones



- Toda excepción hereda de una clase común de Excepción
- Para levantar una excepción, se utiliza “raise”
- Una de las excepciones más comunes en ruby es “Runtime Exception”

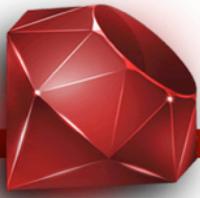
```
f = File.open("testfile")
begin
  # .. proceso
  raise Exception, "FAIL here!!"
rescue OneTypeOfException
  # Manejar un tipo de excepción
rescue AnotherTypeOfException
  retry
else
  # cualquier otra excepcion
ensure
  # siempre ejecutelo
  f.close unless f.nil?
end
```



Tarea

Entrega: 27 de abril

Referencias Bibliográficas



- [http://en.wikipedia.org/wiki/Rake
%28software%29](http://en.wikipedia.org/wiki/Rake_%28software%29)
- [http://en.wikipedia.org/wiki/Ruby
%28programming_language%29](http://en.wikipedia.org/wiki/Ruby_%28programming_language%29)
- <http://www.ruby-doc.org/docs/Tutorial/>
- <http://rubytutorial.wikidot.com/>