



# Introducción a Ruby

Rodrigo Rodriguez  
[rorodr@gmail.com](mailto:rorodr@gmail.com)  
Skype: rarodriguezr

# ¿Qué es Ruby?



- Lenguaje de programación
- Creado en 1995 por Yukihiro “Matz” Matsumoto
- Sintaxis del lenguaje parecido a Perl, Python, and Smalltalk
- Es un lenguaje interpretado, no compilado (como C++, Java)
- Requiere de un interpretador de Ruby (MRI/C-Ruby, jRuby, Rubinius, etc...)

# ¿Por qué Ruby?



- Orientación a Objetos
- Se trata de hacer siempre código legible
- No hay puntos y comas
- Tipado dinámico
- Se puede editar fácilmente objetos y clases lo q facilita la metaprogramación

# Ruby



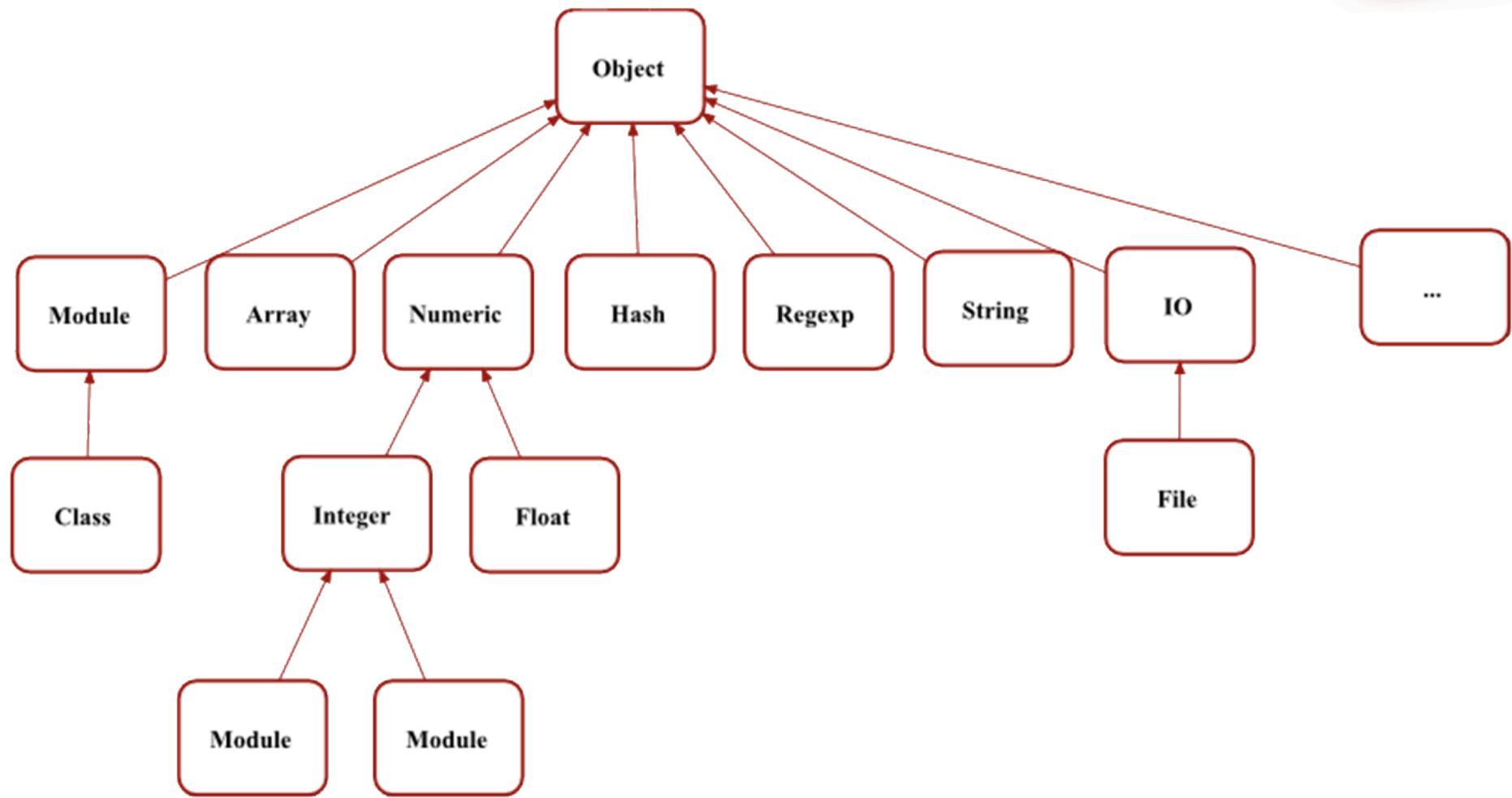
- Ruby es un lenguaje de programación orientado a objetos
- Un objeto es el bloque de desarrollo primordial en el desarrollo de Ruby
- Todo elemento es un objeto (incluyendo métodos)

# Cosas que debemos de saber



- Hay 3 aplicaciones/conceptos importantes que debemos de conocer para la creación de aplicaciones:
  - **IRB(Interactive Ruby Shell)**: Una interfaz de consola que permite la creación de código interactivo
  - **Rake**: Permite la ejecución de tareas predefinidas desde cualquier consola
  - **Gems**: Ruby standard for publishing and managing third party libraries.

# Jerarquía de clases en Ruby



# Variables



- La notación recomendada es comenzar con una letra minúscula o un guión bajo
- Nombres de variables deben de contener únicamente letras, numeros y guión bajo

```
text = 'Hello World!!'  
puts text  
→ "Hello World!!"
```

```
number = 15  
puts number  
→ "15"
```

```
mixted_variable = 10  
puts mixted_variable  
→ "10"
```

```
mixted_variable = "Now I'm a String"  
puts mixted_variable  
→ "Now I'm a String"
```



**Recomendación:** Escribir los nombres de las variables en inglés

# Alcance de las Variables



- **Variables de instancia**, variables que existen mientras que la instancia esté activa.  
`@user_name = "Juan Perez"`
- **Variables de clase**, una variable válida para todas las instancias de una clase.  
`@@configuration_variable = 10`
- **Variables globales**, declaradas con un \$ antes del nombre.
- **Variables locales**, declaradas sin ningún carácter especial en el nombre

# Alcance de las variables



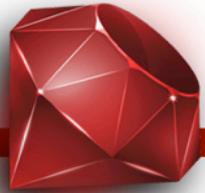
variable_name x_axis thx1138_x	Variable local
@name @point_1 @X @_	Variable de instancia
@@total @@N @@x_pos	Varaible de Clase
\$debug \$CUSTOMER \$_	Variable Global
String MyClass	Class Name
FEET_PER_MILE DEBUG	Constant Name

# Convenciones de nombres



- Métodos y variables: my\_variable, railsTutorial
- Clases, Módulos y Constantes: MyClass, MyModule, ConstantElement

# Valores numéricos



- Los números en Ruby son objetos de la clase Fixnum or Bignum.
- Aquellos valores numéricos que contienen decimales, son objetos de la clase Float
- Ruby puede procesar números muy grandes

```
puts 987**3589
```

```
➔ 402032285964487542274114271545139329938427440508316379008  
06893992433404268716283304349538754855956880564054812519359  
58086 ..... 6151356189647353091129628425845459973867030090027
```

# Algunos Operadores



[ ]	Indices
**	Exponente
* / %	Multiplicación, división, Módulo
+ -	Suma, Resta
« »	Rotación Binaria
&   ^	Operaciones binarias And, Or, Xor
> >= < <=	Comparadores
&&	Boolean ‘And’ y ‘Or’
.. ...	Operadores de rango
= += -=	Métodos de asignación
? :	Operador ternario

# Trabajo en clase 1



1. Escribir una aplicación que muestre el número de minutos que hay en un año de 365 días
2. Escribir una aplicación que genere que la siguiente operación matemática cumpla con el resultado presentado:

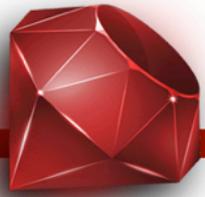
$$(((25 / 45) + 5) * 5) * -2 = 55.56$$

# Strings



- Pueden ser escritos tanto con comillas dobles como con comillas simples
- Es posible interpolar variables en los strings que se declaran con comillas dobles usando la expresión “`a = #{ 21 + 5 }`”
- Otra manera de añadir una variable a un String es usar concatenación: “`hola`” + `variable1`

# Algunos métodos de String



- **reverse**, Invertir todos los caracteres de un string.
- **length**, retorna el total de letras
- **upcase**, todas las letras en mayúscula
- **lowercase**, todas las letras en minúscula
- **capitalize**, Convierte la primera letra de la frase en mayúscula
- **slice**, retorna una sección del texto
- **gsub**, sustituye una sección del string

# Ejemplos de String



- puts "Hello World" → "Hello World"
- puts 'Hello World' → "Hello World"
- puts 'I like' + ' Ruby' → "I like Ruby"
- puts "Hello #{'world'.upcase}"  
→ "Hello WORLD"
- Puts 'Ruby\'s party' → "Ruby's party"
- Puts "Hola" \* 3 → "Hola Hola Hola"

# Trabajo en Clase 2



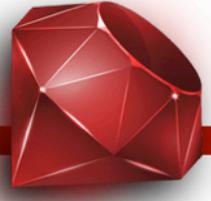
- Imprimir un String que contenga 10 veces las palabras “Hola” y luego concatenarle “Mundo”.
- Considerando el String: “Aprendiendo de Ruby de una manera sencilla” invertir su valor palabra por palabra (en vez de letra por letra)
  - Posiblemente ocupen un poquito de Arrays

# Get User Input



- Ruby permite obtener datos del usuario usando la función “gets”.
- Esta función obtiene toda la información que el usuario escribió, incluyendo incluso el fin de línea.
  - ```
puts "Cual es tu ciudad favorita?"  
STDOUT.flush  
city = gets.chomp  
puts "La ciudad es" + city.capitalize
```

# Trabajo en clase 3



- Escribir una aplicación que solicite por una temperatura en grados Fahrenheit. Esta aplicación utiliza la información brindada por el usuario y calcula el equivalente en grados Celsius. El resultado final se muestra con 2 decimales.
- $\text{Celsius} = \text{Fahrenheit}-32/1,8$

# Métodos



- Declarar métodos en Ruby es simple, solamente se necesita usar “def #{val}”
- El retorno del método puede ser declarado usando el método “return”. Sin embargo, esto no es requerido

```
# Method with one argument.  
def hello(name)  
  puts 'Hola' + name  
  return 'correct'  
end  
# invoking the method  
puts hello ('Team')
```

# “Bang Methods”



- Métodos que terminan con un signo de admiración modifican directamente al objeto.
- Usualmente cada método modificador del objeto tiene una contraparte que crea una copia.

```
# Method sin modificador  
b = a.upcase
```

```
puts b  
puts a
```

```
# Method con modificador  
c = a.upcase!
```

```
puts c  
puts a
```

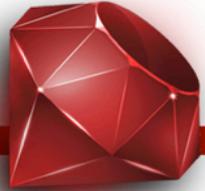
# Alias de métodos



- Crea un nuevo nombre para referirse a un método existente
- Si el método original cambia durante la ejecución, el alias se va a referir al método original

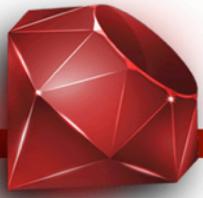
```
def old_method  
  "Este es el método viejo"  
end  
  
alias new_method  
  old_method  
  
def old_method  
  "método mejorado"  
end  
  
puts old_method  
puts new_method
```

# Argumentos en los métodos



- Es posible asignar valores por default usando:
  - `def nombre_metodo(var1='V', var2='M')`
- Es posible declarar un método que recibe una cantidad variable de argumentos utilizando:
  - `def method(*params)`

# Argumentos en los métodos



- Es posible asignar parámetros opcionales usando:
  - `method(param1=2,param2='hello', *param3)`
- Un método también puede recibir bloques de código que se ejecuta en algún momento de la ejecución.
  - Se utiliza “yield” para ejecutar el código del bloque.
  - Se verán más detalles en la clase de bloques

# Trabajo en clase 4



- Refactorizar el código planteado en los trabajos anteriores con el fin de utilizar métodos independientes para cada uno

# Rangos



- Expresan una secuencia numérica
- Las secuencias tienen un valor inicial y un valor final
- Operadores de rangos son “..” And “...”
  - `(1..5).to_a => [1,2,3,4,5]`
  - `(1...6).to_a => [1,2,3,4,5]`

# Arreglos (1/4)



- Los arreglos en Ruby pueden ser declarados creando una nueva instancia de la clase array (Array.new) or asignando “[]” a una variable
- Los arreglos pueden contener objetos de diferentes tipos:
  - [80.5, ‘mango’, [true, false]]
- Se puede generar un arreglo de un texto
  - cities = %w{ Pune Mumbai Bangalore }

# Arreglos (2/4)



- Algunos de los métodos más comunes:
  - Agregar atributos a un arreglo: array << “elem”
  - Cambiar el valor de un arreglo: array[5] = “elem2”
  - Eliminar un elemento: array.delete\_at(5)
  - Eliminar condicional: array.delete\_if { |x| x >= "b" }
  - Buscar un elemento: Array.select { |v| v =~ / [aeiou]/ }

# Arreglos (3/4)



- Algunos métodos adicionales:
  - `array.first`: Retorna el primer elemento
  - `array.last`: retorna el último elemento
  - `array.inspect`: Muestra el arreglo de manera “amigable”
  - `array.join('-')`: une cada uno de los elementos del arreglo con el parámetro establecido

# Arreglos (4/4)



- LA mayoría de casos es necesario iterar todos los elementos, para ello se usa ‘each’ o ‘each\_with\_index’

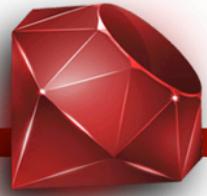
```
cities = ['Paris', 'Miami', 'Sydney']
cities.each do |city|
    puts "Me gustaria ir a #{city}!
end
```

# Trabajo en clase 5: Arreglos



1. Escribir un método que recibe un arreglo que contienen números generados aleatoriamente y retorna la suma de sus elementos
  - **Pista:** `rand(10000)` genera un número aleatorio entre 1 y 10000
2. Crear un método que reciba N cantidad de parámetros, luego itere sobre ellos e imprima en pantalla sus contenidos precedidos por “PARAM:”

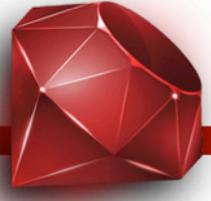
# Condicionales (1/4)



- En un condicional, “nil” y “false” significan falso, todo lo demás es verdadero

```
xyz = rand(12)
if xyz > 4
    puts 'la variable es mayor que 4'
    if xyz == 5
        puts 'La variable es 5'
    elsif xyz > 5 && xyz < 10
        puts 'xyz es mayor que 5 y menor que 10'
    else
        puts 'xyz es mayor que 10'
    end
else
    puts 'La variable no es mayor que 5'
    puts 'Un condicional de una linea' if xyz == 1
end
```

# Condicionales (2/4)



- Los condicionales pueden tener múltiples operandos, los más comunes son:

|                    |                 |
|--------------------|-----------------|
| <code>==</code>    | igual           |
| <code>!=</code>    | Diferente       |
| <code>&gt;=</code> | Mayor o igual a |
| <code>&lt;=</code> | Menor o igual a |
| <code>&gt;</code>  | Mayor que       |
| <code>&lt;</code>  | Menor que       |

# Conditionals (3/4)



- Además de if, se puede usar “case” para elegir entre diversas opciones
- La instrucción “case” puede estar relacionada a un elemento (como un switch en otros lenguajes) o como un sustituto de un if-else.

```
xyz = 10
pair = case
  when xyz % 2 == 0 then true
  when xyz % 2 != 0 then false
End
puts pair
```

```
city = 'rome'
country = case city
  when 'rome' then 'Italy'
  when 'paris' then 'France'
  when 'london' then 'England'
  else 'Unknown'
end
```

# Conditionals (4/4)



- El último condicional que se tiene es el “unless”
- Unless es el opuesto a “IF”: si la condición no es verdadera, ejecute el código, de lo contrario no lo ejecute

```
city = 'Montreal'  
unless city == 'Toronto'  
    puts 'This is not my favorite city in Canada'  
end
```

```
=begin  
If the name is not toronto, execute the block.  
=end
```

# Trabajo 6: Condicionales

---



- Escribir un método que solicita al usuario que digite un año. Luego de procesarlo muestra:
  - El año es biciesto o no
  - El número total de días que tiene el año

**Ayuda:** Años biciestos con aquellos que son divisibles por 4 y no divisible por 100, a no ser que sean también divisibles por 400.

# Trabajo 7: Condicionales

---



- Escribir un método que recibe un argumento (score) y muestra un mensaje según las siguientes reglas:
  - Menor que 50: “Debe de intentarlo nuevamente”
  - Entre 50 y 70: “Casi lo logras!”
  - Entre 70 y 85: “Buen trabajo”
  - Entre 85 y 100: “Excelente trabajo,  
Felicitaciones!”

# Iteradores (1/3)



- Ruby provee muchas maneras de crear ciclos
  - **While**: designado para repetir una tara hasta que la expresión evaluada es *falsa*

```
i = 0
while i < 5 do
    puts i
    i += 1
    break if i == 3
end
```

- **Until**: Itera hasta que la condición sea *verdadera*

```
i = 0
until i < 5 do
    puts i; i += 1
end
```

# Iteradores (2/3)



- **For:** Permite ejecutar una tarea X cantidad de veces

```
for i in 1..8 do  
    puts i  
end
```

- **Times:** Una alternativa conveniente para el iterador “for”

```
10.times { |i| puts i }
```

- **Ranges:** Utilizando los rangos, se pueden crear iteradores

```
(10..30).each { |i| puts i }
```

```
(‘cat’..‘sat’).each { |i| puts i }
```

# Iteradores (3/3)



- **Upto:** Inicia la ejecución en el primer valor y finaliza en el parámetro establecido

```
5.upto(10){|i| puts "hello"}
```

- **Downto:** semejante el upto pero ejecuta de manera descendente

```
15.downto(5){|i| puts i}
```

- **Each:** ejecuta el código para cada elemento en una colección. Este es el iterador más usado en Rails.

# Trabajo 8: Iteradores



- Hacer un método que reciba un valor entero positivo y realice el siguiente cálculo: Si el número es par, divídalo entre 2; si es impar, multiplíquelo por 3 y sume 1. Repita este proceso hasta que el valor final sea 1, imprimiendo en pantalla cada valor

Valor inicial es 5

Siguiente valor es 16

Siguiente valor es 8

Siguiente valor es 4

Siguiente valor es 2

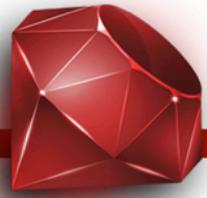
Valor final 1, total de pasos 6

# Trabajo 9: Iteradores



- La oración "A quick brown fox jumps over the lazy dog" contiene todas las letras del alfabeto.
- Esas oraciones son llamadas pangramas
- Escriba la función “missingLetters”, el cual toma un String y retorna todas las letras del alfabeto que falta de utilizar para que la oración sea un pangrama. Es necesario considerar tanto mayúsculas como minúsculas. El retorno del método, debe de ser todas las letras del alfabeto que faltan escritas en minúscula.

# Referencias adicionales



- [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/>