Ruby: un lenguaje dinámico moderno

Esteban Manchado Velázquez
zoso@gulic.org

12 de agosto de 2005

Índice

Índice

Sobre esta charla

- *No* se va a enseñar a programar

Sobre esta charla

- *No* se va a enseñar a programar
- Espero que eso sea un alivio

Sobre esta charla

- *No* se va a enseñar a programar
- Espero que eso sea un alivio
- Presentación a grandes rasgos (falta de tiempo)

Sobre esta charla

- *No* se va a enseñar a programar
- Espero que eso sea un alivio
- Presentación a grandes rasgos (falta de tiempo)
- Más en Rubíes (`http://www.ruby.org.es`)

Índice

Nacimiento de Ruby

- Lo inventó un japonés llamado Yukihiro Matsumoto

Nacimiento de Ruby

- Lo inventó un japonés loco llamado Yukihiro Matsumoto
- Todo el mundo lo conoce como *matz*

Nacimiento de Ruby

- Lo inventó un japonés loco llamado Yukihiro Matsumoto
- Todo el mundo lo conoce como *matz*
- «Ruby» viene de Perl $\longrightarrow$ Pearl

Nacimiento de Ruby

- Lo inventó un japonés loco llamado Yukihiro Matsumoto
- Todo el mundo lo conoce como *matz*
- «Ruby» viene de Perl $\longrightarrow$ Pearl
- *I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python*

Nacimiento de Ruby

- Lo inventó un japonés loco llamado Yukihiro Matsumoto
- Todo el mundo lo conoce como *matz*
- «Ruby» viene de Perl $\longrightarrow$ Pearl
- *I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python*
- *They are focusing on machines. But in fact we need to focus on humans, on how humans care about doing programming or operating the application of the machines. We are the masters. They are the slaves*

Nacimiento de Ruby

- Lo inventó un japonés loco llamado Yukihiro Matsumoto
- Todo el mundo lo conoce como *matz*
- «Ruby» viene de Perl $\longrightarrow$ Pearl
- *I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python*
- *They are focusing on machines. But in fact we need to focus on humans, on how humans care about doing programming or operating the application of the machines. We are the masters. They are the slaves*
- *Don't underestimate the human factor. Even though we are in front of computers, they are media. We are working for human, with human*

Nacimiento de Ruby

- Lo inventó un japonés loco llamado Yukihiro Matsumoto
- Todo el mundo lo conoce como *matz*
- «Ruby» viene de Perl $\longrightarrow$ Pearl
- *I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python*
- *They are focusing on machines. But in fact we need to focus on humans, on how humans care about doing programming or operating the application of the machines. We are the masters. They are the slaves*
- *Don't underestimate the human factor. Even though we are in front of computers, they are media. We are working for human, with human*
- *You want to enjoy life, don't you? If you get your job done quickly and your job is fun, that's good, isn't it? That's the purpose of life, partly. Your life is better*

Características

- «Perl moderno», «Perl orientado a objetos»

Características

- «Perl moderno», «Perl orientado a objetos»
- *Completamente* orientado a objetos (¡hasta `nil`!)

Características

- «Perl moderno», «Perl orientado a objetos»
- *Completamente* orientado a objetos (¡hasta `nil`!)
- Sintaxis limpia, modo poesía, «sufijos» de sentencias

Características

- «Perl moderno», «Perl orientado a objetos»
- *Completamente* orientado a objetos (¡hasta `nil`!)
- Sintaxis limpia, modo poesía, «sufijos» de sentencias
- Sangrado libre, marcas de fin de estructura

Características

- «Perl moderno», «Perl orientado a objetos»
- *Completamente* orientado a objetos (¡hasta `nil`!)
- Sintaxis limpia, modo poesía, «sufijos» de sentencias
- Sangrado libre, marcas de fin de estructura
- Uso de mayúsculas y minúsculas (constantes, variables)

Características

- «Perl moderno», «Perl orientado a objetos»
- *Completamente* orientado a objetos (¡hasta `nil`!)
- Sintaxis limpia, modo poesía, «sufijos» de sentencias
- Sangrado libre, marcas de fin de estructura
- Uso de mayúsculas y minúsculas (constantes, variables)
- Se usan mucho los *bloques* (funciones anónimas)

Características

- «Perl moderno», «Perl orientado a objetos»
- *Completamente* orientado a objetos (¡hasta `nil`!)
- Sintaxis limpia, modo poesía, «sufijos» de sentencias
- Sangrado libre, marcas de fin de estructura
- Uso de mayúsculas y minúsculas (constantes, variables)
- Se usan mucho los *bloques* (funciones anónimas)
- Documentación empotrada

Características

- «Perl moderno», «Perl orientado a objetos»
- *Completamente* orientado a objetos (¡hasta `nil`!)
- Sintaxis limpia, modo poesía, «sufijos» de sentencias
- Sangrado libre, marcas de fin de estructura
- Uso de mayúsculas y minúsculas (constantes, variables)
- Se usan mucho los *bloques* (funciones anónimas)
- Documentación empotrada
- Inmaduro (cambios, pocos módulos *de desarrollo activo*)

Características

- «Perl moderno», «Perl orientado a objetos»
- *Completamente* orientado a objetos (¡hasta `nil`!)
- Sintaxis limpia, modo poesía, «sufijos» de sentencias
- Sangrado libre, marcas de fin de estructura
- Uso de mayúsculas y minúsculas (constantes, variables)
- Se usan mucho los *bloques* (funciones anónimas)
- Documentación empotrada
- Inmaduro (cambios, pocos módulos *de desarrollo activo*)
- Comunidad abierta (refugiados; RoR)

Para aprender...

- Entre Perl y Python en cuanto a integración

Para aprender...

- Entre Perl y Python en cuanto a integración
- Consola interactiva: `irb`

Para aprender...

- Entre Perl y Python en cuanto a integración
- Consola interactiva: `irb`
- Documentación empotrada de referencia: `rdoc`

Para aprender...

- Entre Perl y Python en cuanto a integración
- Consola interactiva: `irb`
- Documentación empotrada de referencia: `rdoc`
- Consulta de documentación, ayuda interactiva: `ri` e `ihelp`

Para aprender...

- Entre Perl y Python en cuanto a integración
- Consola interactiva: `irb`
- Documentación empotrada de referencia: `rdoc`
- Consulta de documentación, ayuda interactiva: `ri` e `ihelp`
- Consulta de documentación en web: `http://www.ruby-doc.org/find/pickaxe/string`

Para aprender...

- Entre Perl y Python en cuanto a integración
- Consola interactiva: `irb`
- Documentación empotrada de referencia: `rdoc`
- Consulta de documentación, ayuda interactiva: `ri` e `ihelp`
- Consulta de documentación en web: `http://www.ruby-doc.org/find/pickaxe/string`
- En el caso de Rails y otros, de moda los *vídeos*

Índice

Micro-intro a la OO

- *Clases* de *objetos*

Micro-intro a la OO

- *Clases* de *objetos*
- Los objetos responden a *métodos*

Micro-intro a la OO

- *Clases* de *objetos*
- Los objetos responden a *métodos*
- Los objetos tienen *atributos*

Micro-intro a la OO

- *Clases* de *objetos*
- Los objetos responden a *métodos*
- Los objetos tienen *atributos*
- Las clases *heredan* de otras

Micro-intro a la OO

- *Clases* de *objetos*
- Los objetos responden a *métodos*
- Los objetos tienen *atributos*
- Las clases *heredan* de otras
- Al menos teóricamente, la OO nos hace natural pensar en términos que *facilitan* reducir el acoplamiento entre conceptos diferentes

Micro-intro a la OO

- *Clases* de *objetos*
- Los objetos responden a *métodos*
- Los objetos tienen *atributos*
- Las clases *heredan* de otras
- Al menos teóricamente, la OO nos hace natural pensar en términos que *facilitan* reducir el acoplamiento entre conceptos diferentes
- **No les culparé si no se lo tragan**

OO en Ruby

- Simple, cómoda de escribir

OO en Ruby

- Simple, cómoda de escribir
- Es la forma natural de resolver los problemas

OO en Ruby

- Simple, cómoda de escribir
- Es la forma natural de resolver los problemas
- Herencia simple

OO en Ruby

- Simple, cómoda de escribir
- Es la forma natural de resolver los problemas
- Herencia simple
- Métodos terminados en «!» y «?»

OO en Ruby

- Simple, cómoda de escribir
- Es la forma natural de resolver los problemas
- Herencia simple
- Métodos terminados en «!» y «?»
- Métodos especiales «=» para caramelos sintácticos

OO en Ruby

- Simple, cómoda de escribir
- Es la forma natural de resolver los problemas
- Herencia simple
- Métodos terminados en «!» y «?»
- Métodos especiales «=» para caramelos sintácticos
- *No existen* los atributos (desde fuera)

OO en Ruby

- Simple, cómoda de escribir
- Es la forma natural de resolver los problemas
- Herencia simple
- Métodos terminados en «!» y «?»
- Métodos especiales «=» para caramelos sintácticos
- *No existen* los atributos (desde fuera)
- Se usa «@» y «@@» para los atributos de objeto/clase

OO en Ruby

- Simple, cómoda de escribir
- Es la forma natural de resolver los problemas
- Herencia simple
- Métodos terminados en «!» y «?»
- Métodos especiales «=» para caramelos sintácticos
- *No existen* los atributos (desde fuera)
- Se usa «@» y «@@» para los atributos de objeto/clase
- Se puede escribir en estilo *no* OO, pero en realidad es OO

Estructuras

- Pocas, en parte por los bloques

Estructuras

- Pocas, en parte por los bloques
- `if`, `case`, `while`, `loop`, `for`

Estructuras

- Pocas, en parte por los bloques
- `if`, `case`, `while`, `loop`, `for`
- `if` tiene `elsif` aparte de `else`, y devuelve un valor

Estructuras

- Pocas, en parte por los bloques
- `if`, `case`, `while`, `loop`, `for`
- `if` tiene `elsif` aparte de `else`, y devuelve un valor
- `if` tiene `then` optativo (sintaxis de una línea)

Estructuras

- Pocas, en parte por los bloques
- `if`, `case`, `while`, `loop`, `for`
- `if` tiene `elsif` aparte de `else`, y devuelve un valor
- `if` tiene `then` optativo (sintaxis de una línea)
- `case` tiene una forma curiosa de comparar (operador `===` *de lo que va en el* `when`)

Estructuras

- Pocas, en parte por los bloques
- `if`, `case`, `while`, `loop`, `for`
- `if` tiene `elsif` aparte de `else`, y devuelve un valor
- `if` tiene `then` optativo (sintaxis de una línea)
- `case` tiene una forma curiosa de comparar (operador `===` *de lo que va en el* `when`)
- `case` devuelve un valor

Estructuras

- Pocas, en parte por los bloques
- `if`, `case`, `while`, `loop`, `for`
- `if` tiene `elsif` aparte de `else`, y devuelve un valor
- `if` tiene `then` optativo (sintaxis de una línea)
- `case` tiene una forma curiosa de comparar (operador `===` *de lo que va en el* `when`)
- `case` devuelve un valor
- Las tres últimas rara vez se usan

Ejemplo

```
if 0 then puts "Sorpresa" else puts "0 es falso" end

if 0
   puts "No te enteras..."
else
   puts "Solamente false y nil son falsos"
end
```

Ejemplo

```
if 0 then puts "Sorpresa" else puts "0 es falso" end

if 0
   puts "No te enteras..."
else
   puts "Solamente false y nil son falsos"
end
```

Ejemplo

```
if 0 then puts "Sorpresa" else puts "0 es falso" end

if 0
   puts "No te enteras..."
else
   puts "Solamente false y nil son falsos"
end
```

Más ejemplos

```
case 'pepito'
   when /pep/
      puts "Guardiola"
   when Integer
      puts "Un entero"
   when 'a' .. 'z'
      puts "Una sola letra"
end
```

Más ejemplos

```
case 'pepito'
   when /pep/
      puts "Guardiola"
   when Integer
      puts "Un entero"
   when 'a' .. 'z'
      puts "Una sola letra"
end
```

Más ejemplos

```
case 'pepito'
   when /pep/
      puts "Guardiola"
   when Integer
      puts "Un entero"
   when 'a' .. 'z'
      puts "Una sola letra"
end
```

Más ejemplos

```
case 'pepito'
   when /pep/
      puts "Guardiola"
   when Integer
      puts "Un entero"
   when 'a' .. 'z'
      puts "Una sola letra"
end
```

Más ejemplos todavía

```
a = 0
while a < 10 do a += 1 end

loop do
   a += 1
   puts "Bucle manual y aburrido"
   break if a > 20
end

for i in 35..40 do puts "Mejor, pero no" end
```

Bucles estilo Ruby

```
a = 0
while a < 10 do a += 1 end

loop do
   a += 1
   puts "Bucle manual y aburrido"
   break if a > 20
end

for i in 35..40 do puts "Mejor, pero no" end
```

Bucles estilo Ruby

```
0.upto(9) do puts "Mejor" end

loop do
   a += 1
   puts "Bucle manual y aburrido"
   break if a > 20
end

for i in 35..40 do puts "Mejor, pero no" end
```

Bucles estilo Ruby

```
0.upto(9) do puts "Mejor" end
```

```
10.times do puts "Bucle rubyano" end
```

```
for i in 35..40 do puts "Mejor, pero no" end
```

Bucles estilo Ruby

```
0.upto(9) do puts "Mejor" end
```

```
10.times do puts "Bucle rubyano" end
```

```
(35..40).each do |i| puts "Te saliste #{i}" end
```

Bloques e iteradores

- La base del lenguaje

Bloques e iteradores

- La base del lenguaje
- Varios usos típicos:

Bloques e iteradores

- La base del lenguaje
- Varios usos típicos:
    - Bucles (iteradores)

Bloques e iteradores

- La base del lenguaje
- Varios usos típicos:
  - Bucles (iteradores)
  - Guardar funciones anónimas (para más tarde)

Bloques e iteradores

- La base del lenguaje
- Varios usos típicos:
  - Bucles (iteradores)
  - Guardar funciones anónimas (para más tarde)
  - Ejecutar algo con un recurso (gestión automática)

Bloques e iteradores

- La base del lenguaje
- Varios usos típicos:
  - Bucles (iteradores)
  - Guardar funciones anónimas (para más tarde)
  - Ejecutar algo con un recurso (gestión automática)
  - Inicialización de atributos

Bloques e iteradores

- La base del lenguaje
- Varios usos típicos:
  - Bucles (iteradores)
  - Guardar funciones anónimas (para más tarde)
  - Ejecutar algo con un recurso (gestión automática)
  - Inicialización de atributos
- Se pueden convertir en objetos `Proc` para pasarse

Ejemplo de bloques

```
[1, 2, 3].each {|i| puts i}
```

Ejemplo de bloques

```
[1, 2, 3].each {|i| puts i}

button.onClick do |event, obj, data|
  puts "Me han pulsado"
end
```

Ejemplo de bloques

```
[1, 2, 3].each {|i| puts i}

button.onClick do |event, obj, data|
  puts "Me han pulsado"
end

File.open("foo") do |f|
  # Hacemos algo con f, que se cierra solo al final
end
```

Ejemplo de bloques

```
[1, 2, 3].each {|i| puts i}

button.onClick do |event, obj, data|
  puts "Me han pulsado"
end

File.open("foo") do |f|
  # Hacemos algo con f, que se cierra solo al final
end

UnaClase.new do |o|
  o.attr1 = ; o.attr2 = 30
end
```

Ahora lo ves, ahora no lo ves

- Ruby es *muy* dinámico

Ahora lo ves, ahora no lo ves

- Ruby es *muy* dinámico
- Casi todo se puede redefinir (¡incluso la base!)

Ahora lo ves, ahora no lo ves

- Ruby es *muy* dinámico
- Casi todo se puede redefinir (¡incluso la base!)
- Las definiciones de clase se «ejecutan»...

Ahora lo ves, ahora no lo ves

- Ruby es *muy* dinámico
- Casi todo se puede redefinir (¡incluso la base!)
- Las definiciones de clase se «ejecutan»...
- ...así que podemos «ejecutar» cosas al definir clases

Ahora lo ves, ahora no lo ves

- Ruby es *muy* dinámico
- Casi todo se puede redefinir (¡incluso la base!)
- Las definiciones de clase se «ejecutan»...
- ...así que podemos «ejecutar» cosas al definir clases
- `alias_method`, `attr_reader`, `protected`, `public`, `module_function`...

Ahora lo ves, ahora no lo ves

- Ruby es *muy* dinámico
- Casi todo se puede redefinir (¡incluso la base!)
- Las definiciones de clase se «ejecutan»...
- ...así que podemos «ejecutar» cosas al definir clases
- `alias_method`, `attr_reader`, `protected`, `public`, `module_function`...
- Declarar una clase es definir una nueva *constante* (mayúsculas, *Remember, Sammy Jenkins?*)

Ahora lo ves, ahora no lo ves

- Ruby es *muy* dinámico
- Casi todo se puede redefinir (¡incluso la base!)
- Las definiciones de clase se «ejecutan»...
- ...así que podemos «ejecutar» cosas al definir clases
- `alias_method`, `attr_reader`, `protected`, `public`, `module_function`...
- Declarar una clase es definir una nueva *constante* (mayúsculas, *Remember, Sammy Jenkins?*)
- Más sobre esto en los extras del DVD

Ejemplo

```
class Apitijander
   attr_reader :foo          # Igual a def foo; @foo; end
   attr_accessor :bar        # Tambien crea bar=

   protected                 # Esto es un metodo
   def metodo_protegido
   end
end

module UnModulo
   def f1
   end

   module_function :f1
end
```

Ejemplo

```
class Apitijander
    attr_reader :foo          # Igual a def foo; @foo; end
    attr_accessor :bar        # Tambien crea bar=

    protected                 # Esto es un metodo
    def metodo_protegido
    end
end

module UnModulo
    def f1
    end

    module_function :f1
end
```

Ejemplo

```
class Apitijander
   attr_reader :foo          # Igual a def foo; @foo; end
   attr_accessor :bar        # Tambien crea bar=

   protected                 # Esto es un metodo
   def metodo_protegido
   end
end

module UnModulo
   def f1
   end

   module_function :f1
end
```

Ejemplo

```
class Apitijander
    attr_reader :foo          # Igual a def foo; @foo; end
    attr_accessor :bar        # Tambien crea bar=

    protected                 # Esto es un metodo
    def metodo_protegido
    end
end

module UnModulo
    def f1
    end

    module_function :f1
end
```

Otro ejemplo

```
class Gromenauer
end

g = Gromenauer.new
if g.class == Gromenauer
   class String
      def ataque_al_corazon
         puts "Ya te botaste"
      end
   end
end

if "".respond_to?(:ataque_al_corazon)
   "Cuidado con el".ataque_al_corazon
end
```

Otro ejemplo

```
class Gromenauer
end

g = Gromenauer.new
if g.class == Gromenauer
   class String
      def ataque_al_corazon
         puts "Ya te botaste"
      end
   end
end

if "".respond_to?(:ataque_al_corazon)
   "Cuidado con el".ataque_al_corazon
end
```

Otro ejemplo

```
class Gromenauer
end

g = Gromenauer.new
if g.class == Gromenauer
   class String
      def ataque_al_corazon
         puts "Ya te botaste"
      end
   end
end

if "".respond_to?(:ataque_al_corazon)
   "Cuidado con el".ataque_al_corazon
end
```

Otro ejemplo

```
class Gromenauer
end

g = Gromenauer.new
if g.class == Gromenauer
   class String
      def ataque_al_corazon
         puts "Ya te botaste"
      end
   end
end

if "".respond_to?(:ataque_al_corazon)
   "Cuidado con el".ataque_al_corazon
end
```

Otro ejemplo

```
class Gromenauer
end

g = Gromenauer.new
if g.class == Gromenauer
   class String
      def ataque_al_corazon
         puts "Ya te botaste"
      end
   end
end

if "".respond_to?(:ataque_al_corazon)
   "Cuidado con el".ataque_al_corazon
end
```

Ejemplo más real

```
# Redefine funciones para que se ejecuten una sola vez
# Se usa como once :unmetodo, :otrometodo
def Clase.once(*ids)
  for id in ids
    module_eval <<-"end_eval"
      alias_method :__#{id.to_i}__, #{id.inspect}
      def #{id.id2name}(*args, &block)
        def self.#{id.id2name}(*args, &block)
          @__#{id.to_i}__
        end
        @__#{id.to_i}__ = __#{id.to_i}__(*args, &block)
      end
    end_eval
  end
end
```

Descanso

ツテボミヴス

Descanso

ツテボミヴス

Índice

Módulos

- «Declaran» nuevos espacios de nombres

Módulos

- «Declaran» nuevos espacios de nombres
- Son explícitos, con la sintaxis `module Foo ... end`

Módulos

- «Declaran» nuevos espacios de nombres
- Son explícitos, con la sintaxis `module Foo ... end`
- La orden `require` **no** los incluye en un espacio nuevo

Módulos

- «Declaran» nuevos espacios de nombres
- Son explícitos, con la sintaxis `module Foo ... end`
- La orden `require` **no** los incluye en un espacio nuevo
- Esto permite redefinir el contexto actual, p.ej.

Módulos

- «Declaran» nuevos espacios de nombres
- Son explícitos, con la sintaxis `module Foo ... end`
- La orden `require` **no** los incluye en un espacio nuevo
- Esto permite redefinir el contexto actual, p.ej.
- Como las clases, pero no se pueden crear *ejemplares* de módulos

Módulos

- «Declaran» nuevos espacios de nombres
- Son explícitos, con la sintaxis `module Foo ... end`
- La orden `require` **no** los incluye en un espacio nuevo
- Esto permite redefinir el contexto actual, p.ej.
- Como las clases, pero no se pueden crear *ejemplares* de módulos
- Sirven también para la técnica *Mix-in*

Módulos

- «Declaran» nuevos espacios de nombres
- Son explícitos, con la sintaxis `module Foo ... end`
- La orden `require` **no** los incluye en un espacio nuevo
- Esto permite redefinir el contexto actual, p.ej.
- Como las clases, pero no se pueden crear *ejemplares* de módulos
- Sirven también para la técnica *Mix-in*
- Las funciones en módulos se declaran como `def Foo.funcion` (o con `module_function`)

Ejemplo de módulos

```
module Foo
  class Bar; end
end

f = Foo::Bar.new

class Foo::Bar
  def dia_aciago; puts "Asi hago algo"; end
end

f.dia_aciago          # "Asi hago algo"
```

Ejemplo de módulos

```
module Foo
  class Bar; end
end

f = Foo::Bar.new

class Foo::Bar
  def dia_aciago; puts "Asi hago algo"; end
end

f.dia_aciago          # "Asi hago algo"
```

Ejemplo de módulos

```
module Foo
  class Bar; end
end

f = Foo::Bar.new

class Foo::Bar
  def dia_aciago; puts "Asi hago algo"; end
end

f.dia_aciago          # "Asi hago algo"
```

Ejemplo de módulos

```
module Foo
  class Bar; end
end

f = Foo::Bar.new

class Foo::Bar
  def dia_aciago; puts "Asi hago algo"; end
end

f.dia_aciago          # "Asi hago algo"
```

Ejemplo de módulos

```
module Foo
  class Bar; end
end

f = Foo::Bar.new

class Foo::Bar
  def dia_aciago; puts "Asi hago algo"; end
end

f.dia_aciago          # "Asi hago algo"
```

Funciones en módulos

```
module Foo
  def met_mixin; "mixin"; end
  def Foo.met_normal; "normal"; end
  def met_normal2; "normal2"; end
  module_function :met_normal2
end

Foo.met_mixin      # Lanza NoMethodError
Foo.met_normal     # "normal"
Foo.met_normal2    # "normal2"
```

Funciones en módulos

```
module Foo
  def met_mixin; "mixin"; end
  def Foo.met_normal; "normal"; end
  def met_normal2; "normal2"; end
  module_function :met_normal2
end

Foo.met_mixin       # Lanza NoMethodError
Foo.met_normal      # "normal"
Foo.met_normal2     # "normal2"
```

Funciones en módulos

```
module Foo
  def met_mixin; "mixin"; end
  def Foo.met_normal; "normal"; end
  def met_normal2; "normal2"; end
  module_function :met_normal2
end

Foo.met_mixin      # Lanza NoMethodError
Foo.met_normal     # "normal"
Foo.met_normal2    # "normal2"
```

Funciones en módulos

```
module Foo
  def met_mixin; "mixin"; end
  def Foo.met_normal; "normal"; end
  def met_normal2; "normal2"; end
  module_function :met_normal2
end

Foo.met_mixin       # Lanza NoMethodError
Foo.met_normal      # "normal"
Foo.met_normal2     # "normal2"
```

Símbolos, ¿ser o estar?

- Cosa «rara» de Ruby

Símbolos, ¿ser o estar?

- Cosa «rara» de Ruby
- Ninguna *funcionalidad* adicional, mera «forma de hablar»

Símbolos, ¿ser o estar?

- Cosa «rara» de Ruby
- Ninguna *funcionalidad* adicional, mera «forma de hablar»
- Parecidos a ristras inmutables

Símbolos, ¿ser o estar?

- Cosa «rara» de Ruby
- Ninguna *funcionalidad* adicional, mera «forma de hablar»
- Parecidos a ristras inmutables
- Sintaxis $\longrightarrow$ `:simbolo`

Símbolos, ¿ser o estar?

- Cosa «rara» de Ruby
- Ninguna *funcionalidad* adicional, mera «forma de hablar»
- Parecidos a ristras inmutables
- Sintaxis $\longrightarrow$ `:simbolo`
- Ayudan a identificar la intención

Símbolos, ¿ser o estar?

- Cosa «rara» de Ruby
- Ninguna *funcionalidad* adicional, mera «forma de hablar»
- Parecidos a ristras inmutables
- Sintaxis $\longrightarrow$ `:simbolo`
- Ayudan a identificar la intención
- No hay equivalente en otros lenguajes populares, en los que se usarían ristras (¿*be* = «ser» o «estar»?)

Ejemplos de símbolos

```
File.open('ruta')
obj.send(:metodo)       # Aunque vale obj.send('metodo')

var['clave'] = 'valor'
var[:opcion] = 'valor'

find(:conds => ["1 = :foo", :foo => 'bar'],
            :limit => 3, :order_by => "uncampo DESC")
set_table_name 'una_tabla'
valides_presence_of :uncampo
```

Ejemplos de símbolos

```
File.open('ruta')
obj.send(:metodo)       # Aunque vale obj.send('metodo')

var['clave'] = 'valor'
var[:opcion] = 'valor'

find(:conds => ["1 = :foo", :foo => 'bar'],
            :limit => 3, :order_by => "uncampo DESC")
set_table_name 'una_tabla'
valides_presence_of :uncampo
```

Ejemplos de símbolos

```
File.open('ruta')
obj.send(:metodo)       # Aunque vale obj.send('metodo')

var['clave'] = 'valor'
var[:opcion] = 'valor'

find(:conds => ["1 = :foo", :foo => 'bar'],
            :limit => 3, :order_by => "uncampo DESC")
set_table_name 'una_tabla'
valides_presence_of :uncampo
```

Ejemplos de símbolos

```
File.open('ruta')
obj.send(:metodo)       # Aunque vale obj.send('metodo')

var['clave'] = 'valor'
var[:opcion] = 'valor'

find(:conds => ["1 = :foo", :foo => 'bar'],
             :limit => 3, :order_by => "uncampo DESC")
set_table_name 'una_tabla'
valides_presence_of :uncampo
```

Ejemplos de símbolos

```
File.open('ruta')
obj.send(:metodo)       # Aunque vale obj.send('metodo')

var['clave'] = 'valor'
var[:opcion] = 'valor'

find(:conds => ["1 = :foo", :foo => 'bar'],
            :limit => 3, :order_by => "uncampo DESC")
set_table_name 'una_tabla'
valides_presence_of :uncampo
```

Ejemplos de símbolos

```
File.open('ruta')
obj.send(:metodo)       # Aunque vale obj.send('metodo')

var['clave'] = 'valor'
var[:opcion] = 'valor'

find(:conds => ["1 = :foo", :foo => 'bar'],
             :limit => 3, :order_by => "uncampo DESC")
set_table_name 'una_tabla'
valides_presence_of :uncampo
```

Ejemplos de símbolos

```
File.open('ruta')
obj.send(:metodo)       # Aunque vale obj.send('metodo')

var['clave'] = 'valor'
var[:opcion] = 'valor'

find(:conds => ["1 = :foo", :foo => 'bar'],
             :limit => 3, :order_by => "uncampo DESC")
set_table_name 'una_tabla'
valides_presence_of :uncampo
```

Excepciones

- Ningún detalle especial/sorprendente

Excepciones

- Ningún detalle especial/sorprendente
- Clases, que pueden heredar (la base es `Exception`)

Excepciones

- Ningún detalle especial/sorprendente
- Clases, que pueden heredar (la base es `Exception`)
- Por convención terminan en `Error`

Excepciones

- Ningún detalle especial/sorprendente
- Clases, que pueden heredar (la base es `Exception`)
- Por convención terminan en `Error`
- Se cazan sólo las herederas de `StandardError`, si no especificamos

Excepciones

- Ningún detalle especial/sorprendente
- Clases, que pueden heredar (la base es `Exception`)
- Por convención terminan en `Error`
- Se cazan sólo las herederas de `StandardError`, si no especificamos
- Si lanzamos una sin especificar, se lanza `RuntimeError`

Excepciones

- Ningún detalle especial/sorprendente
- Clases, que pueden heredar (la base es `Exception`)
- Por convención terminan en `Error`
- Se cazan sólo las herederas de `StandardError`, si no especificamos
- Si lanzamos una sin especificar, se lanza `RuntimeError`
- Se lanzan con `raise`

Excepciones

- Ningún detalle especial/sorprendente
- Clases, que pueden heredar (la base es `Exception`)
- Por convención terminan en `Error`
- Se cazan sólo las herederas de `StandardError`, si no especificamos
- Si lanzamos una sin especificar, se lanza `RuntimeError`
- Se lanzan con `raise`
- Se capturan con bloques `begin`/`rescue`/`end`

Excepciones

- Ningún detalle especial/sorprendente
- Clases, que pueden heredar (la base es `Exception`)
- Por convención terminan en `Error`
- Se cazan sólo las herederas de `StandardError`, si no especificamos
- Si lanzamos una sin especificar, se lanza `RuntimeError`
- Se lanzan con `raise`
- Se capturan con bloques `begin`/`rescue`/`end`
- Bloque `ensure`, para ejecutar algo *siempre*

Ejemplos de excepciones

```
begin
  raise                    # Como RuntimeError, ""
  raise "Mensaje"          # Como RuntimeError
  raise AlgunaExcepcionError, "Mensaje"
  raise ArgumentError, "Name too big", caller[1..-1]
rescue SyntaxError, NameError => boom
  puts "Error de sintaxis: #{boom}"
rescue AlgunaExcepcionError
  puts "Estilo Perl: #{$!}"
rescue => e                  # Como StandardError
  print "Error: #{e}"
else
  puts "Esto no se imprime"
ensure
  puts "Se ejecuta siempre"
end
```

Ejemplos de excepciones

```
begin
  raise                       # Como RuntimeError, ""
  raise "Mensaje"             # Como RuntimeError
  raise AlgunaExcepcionError, "Mensaje"
  raise ArgumentError, "Name too big", caller[1..-1]
rescue SyntaxError, NameError => boom
  puts "Error de sintaxis: #{boom}"
rescue AlgunaExcepcionError
  puts "Estilo Perl: #{$!}"
rescue => e                   # Como StandardError
  print "Error: #{e}"
else
  puts "Esto no se imprime"
ensure
  puts "Se ejecuta siempre"
end
```

Ejemplos de excepciones

```
begin
  raise                    # Como RuntimeError, ""
  raise "Mensaje"          # Como RuntimeError
  raise AlgunaExcepcionError, "Mensaje"
  raise ArgumentError, "Name too big", caller[1..-1]
rescue SyntaxError, NameError => boom
  puts "Error de sintaxis: #{boom}"
rescue AlgunaExcepcionError
  puts "Estilo Perl: #{$!}"
rescue => e                  # Como StandardError
  print "Error: #{e}"
else
  puts "Esto no se imprime"
ensure
  puts "Se ejecuta siempre"
end
```

Ejemplos de excepciones

```
begin
  raise                    # Como RuntimeError, ""
  raise "Mensaje"          # Como RuntimeError
  raise AlgunaExcepcionError, "Mensaje"
  raise ArgumentError, "Name too big", caller[1..-1]
rescue SyntaxError, NameError => boom
  puts "Error de sintaxis: #{boom}"
rescue AlgunaExcepcionError
  puts "Estilo Perl: #{$!}"
rescue => e                  # Como StandardError
  print "Error: #{e}"
else
  puts "Esto no se imprime"
ensure
  puts "Se ejecuta siempre"
end
```

Ejemplos de excepciones

```
begin
  raise                    # Como RuntimeError, ""
  raise "Mensaje"          # Como RuntimeError
  raise AlgunaExcepcionError, "Mensaje"
  raise ArgumentError, "Name too big", caller[1..-1]
rescue SyntaxError, NameError => boom
  puts "Error de sintaxis: #{boom}"
rescue AlgunaExcepcionError
  puts "Estilo Perl: #{$!}"
rescue => e                  # Como StandardError
  print "Error: #{e}"
else
  puts "Esto no se imprime"
ensure
  puts "Se ejecuta siempre"
end
```

Ejemplos de excepciones

```
begin
  raise                    # Como RuntimeError, ""
  raise "Mensaje"          # Como RuntimeError
  raise AlgunaExcepcionError, "Mensaje"
  raise ArgumentError, "Name too big", caller[1..-1]
rescue SyntaxError, NameError => boom
  puts "Error de sintaxis: #{boom}"
rescue AlgunaExcepcionError
  puts "Estilo Perl: #{$!}"
rescue => e                  # Como StandardError
  print "Error: #{e}"
else
  puts "Esto no se imprime"
ensure
  puts "Se ejecuta siempre"
end
```

Ejemplos de excepciones

```
begin
  raise                    # Como RuntimeError, ""
  raise "Mensaje"          # Como RuntimeError
  raise AlgunaExcepcionError, "Mensaje"
  raise ArgumentError, "Name too big", caller[1..-1]
rescue SyntaxError, NameError => boom
  puts "Error de sintaxis: #{boom}"
rescue AlgunaExcepcionError
  puts "Estilo Perl: #{$!}"
rescue => e                  # Como StandardError
  print "Error: #{e}"
else
  puts "Esto no se imprime"
ensure
  puts "Se ejecuta siempre"
end
```

Ejemplos de excepciones

```
begin
  raise                    # Como RuntimeError, ""
  raise "Mensaje"          # Como RuntimeError
  raise AlgunaExcepcionError, "Mensaje"
  raise ArgumentError, "Name too big", caller[1..-1]
rescue SyntaxError, NameError => boom
  puts "Error de sintaxis: #{boom}"
rescue AlgunaExcepcionError
  puts "Estilo Perl: #{$!}"
rescue => e                  # Como StandardError
  print "Error: #{e}"
else
  puts "Esto no se imprime"
ensure
  puts "Se ejecuta siempre"
end
```

Ejemplos de excepciones

```
begin
  raise                    # Como RuntimeError, ""
  raise "Mensaje"          # Como RuntimeError
  raise AlgunaExcepcionError, "Mensaje"
  raise ArgumentError, "Name too big", caller[1..-1]
rescue SyntaxError, NameError => boom
  puts "Error de sintaxis: #{boom}"
rescue AlgunaExcepcionError
  puts "Estilo Perl: #{$!}"
rescue => e                  # Como StandardError
  print "Error: #{e}"
else
  puts "Esto no se imprime"
ensure
  puts "Se ejecuta siempre"
end
```

Ejemplos de excepciones

```
begin
  raise                    # Como RuntimeError, ""
  raise "Mensaje"          # Como RuntimeError
  raise AlgunaExcepcionError, "Mensaje"
  raise ArgumentError, "Name too big", caller[1..-1]
rescue SyntaxError, NameError => boom
  puts "Error de sintaxis: #{boom}"
rescue AlgunaExcepcionError
  puts "Estilo Perl: #{$!}"
rescue => e                  # Como StandardError
  print "Error: #{e}"
else
  puts "Esto no se imprime"
ensure
  puts "Se ejecuta siempre"
end
```

Expresiones regulares

- «Empotradas» en la sintaxis

Expresiones regulares

- «Empotradas» en la sintaxis
- Búsqueda con `expresión =~/expreg/`

Expresiones regulares

- «Empotradas» en la sintaxis
- Búsqueda con `expresión =~/expreg/`
- Sustitución con `expresión.gsub(/ex(pr)eg/, '\1')`

Expresiones regulares

- «Empotradas» en la sintaxis
- Búsqueda con `expresión =~/expreg/`
- Sustitución con `expresión.gsub(/ex(pr)eg/, '\1')`
- También con `expresión.gsub(/ex(pr)eg/) {|p| $1}`

Expresiones regulares

- «Empotradas» en la sintaxis
- Búsqueda con `expresión =~/expreg/`
- Sustitución con `expresión.gsub(/ex(pr)eg/, '\1')`
- También con `expresión.gsub(/ex(pr)eg/) {|p| $1}`
- En el fondo, caramelos sintácticos (completamente OO)

Expresiones regulares

- «Empotradas» en la sintaxis
- Búsqueda con `expresión =~/expreg/`
- Sustitución con `expresión.gsub(/ex(pr)eg/, '\1')`
- También con `expresión.gsub(/ex(pr)eg/) {|p| $1}`
- En el fondo, caramelos sintácticos (completamente OO)
- Clases `Regexp`, `MatchData`

Ejemplos de expresiones regulares

```
if "jander" =~ /m/
  puts "Tiene una m"
else
  puts "No tiene m"
end

"jander".gsub(/nde/, 'la')

vars = { 'var1' => 'val1', 'var2' => 'val2' }
plantilla = "var1 =%(var1), otravar =%(otravar)"
plantilla.gsub(/%([a-z0-9_]+)/) do |s|
  vars.has_key?($1) ? vars[$1] : s
end
```

Ejemplos de expresiones regulares

```
if "jander" =~ /m/
  puts "Tiene una m"
else
  puts "No tiene m"
end

"jander".gsub(/nde/, 'la')

vars = { 'var1' => 'val1', 'var2' => 'val2' }
plantilla = "var1 =%(var1), otravar =%(otravar)"
plantilla.gsub(/%([a-z0-9_]+)/) do |s|
  vars.has_key?($1) ? vars[$1] : s
end
```

Ejemplos de expresiones regulares

```
if "jander" =~ /m/
  puts "Tiene una m"
else
  puts "No tiene m"
end

"jander".gsub(/nde/, 'la')

vars = { 'var1' => 'val1', 'var2' => 'val2' }
plantilla = "var1 =%(var1), otravar =%(otravar)"
plantilla.gsub(/%([a-z0-9_]+)/) do |s|
  vars.has_key?($1) ? vars[$1] : s
end
```

Ejemplos de expresiones regulares

```
if "jander" =~ /m/
  puts "Tiene una m"
else
  puts "No tiene m"
end

"jander".gsub(/nde/, 'la')

vars = { 'var1' => 'val1', 'var2' => 'val2' }
plantilla = "var1 =%(var1), otravar =%(otravar)"
plantilla.gsub(/%([a-z0-9_]+)/) do |s|
  vars.has_key?($1) ? vars[$1] : s
end
```

Rangos

- Tres usos básicos: series, condiciones e intervalos

Rangos

- Tres usos básicos: series, condiciones e intervalos
- Series: como listas, pero más eficientes

Rangos

- Tres usos básicos: series, condiciones e intervalos
- Series: como listas, pero más eficientes
- Condiciones: guardan internamente su estado, para usarlas como condición

Rangos

- Tres usos básicos: series, condiciones e intervalos
- Series: como listas, pero más eficientes
- Condiciones: guardan internamente su estado, para usarlas como condición
- Intervalos: comprobar si un valor entra dentro de un intervalo

Ejemplos de rangos

```
('a'..'z').each do |letra|
   if (letra == 'h') .. (letra == 'v')
      puts "Ni al principio ni al final"
   end
end

File.readlines('numeros').each do |n|
  puts n if n =~ /tres/ .. n =~ /seis/
end

if (1 .. 5) === 3 then puts "Entre los primeros"
```

Ejemplos de rangos

```
('a'..'z').each do |letra|
   if (letra == 'h') .. (letra == 'v')
      puts "Ni al principio ni al final"
   end
end

File.readlines('numeros').each do |n|
  puts n if n =~ /tres/ .. n =~ /seis/
end

if (1 .. 5) === 3 then puts "Entre los primeros"
```

Ejemplos de rangos

```
('a'..'z').each do |letra|
   if (letra == 'h') .. (letra == 'v')
      puts "Ni al principio ni al final"
   end
end

File.readlines('numeros').each do |n|
  puts n if n =~ /tres/ .. n =~ /seis/
end

if (1 .. 5) === 3 then puts "Entre los primeros"
```

Ejemplos de rangos

```
('a'..'z').each do |letra|
   if (letra == 'h') .. (letra == 'v')
      puts "Ni al principio ni al final"
   end
end

File.readlines('numeros').each do |n|
  puts n if n =~ /tres/ .. n =~ /seis/
end

if (1 .. 5) === 3 then puts "Entre los primeros"
```

Más ejemplos de rangos

```
case foo
when 1..3                    # Igual que (1..3) === foo
  puts "Mini"
when 4..6
  puts "Mediano"
when 7..9
  puts "Grande"
end
```

Índice

Qué

- La herencia en Ruby *no es* múltiple

Qué

- La herencia en Ruby *no es* múltiple
- Tampoco hay plantillas al estilo C++ (tipos dinámicos)

Qué

- La herencia en Ruby *no es* múltiple
- Tampoco hay plantillas al estilo C++ (tipos dinámicos)
- Es conveniente compartir funcionalidad genérica

Qué

- La herencia en Ruby *no es* múltiple
- Tampoco hay plantillas al estilo C++ (tipos dinámicos)
- Es conveniente compartir funcionalidad genérica
- O bien obligar a que ciertas clases compartan ciertos métodos

Qué

- La herencia en Ruby *no es* múltiple
- Tampoco hay plantillas al estilo C++ (tipos dinámicos)
- Es conveniente compartir funcionalidad genérica
- O bien obligar a que ciertas clases compartan ciertos métodos
- La solución en Ruby es el *Mixin*

Cómo

- Se añaden métodos a un *módulo*, y luego se incluyen en las clases

Cómo

- Se añaden métodos a un *módulo*, y luego se incluyen en las clases
- Los métodos se añaden como si fueran del *ejemplar*, no de la *clase* (no se pueden llamar directamente)

Cómo

- Se añaden métodos a un *módulo*, y luego se incluyen en las clases
- Los métodos se añaden como si fueran del *ejemplar*, no de la *clase* (no se pueden llamar directamente)
- En la clase, se hace un `include Modulo`...

Cómo

- Se añaden métodos a un *módulo*, y luego se incluyen en las clases
- Los métodos se añaden como si fueran del *ejemplar*, no de la *clase* (no se pueden llamar directamente)
- En la clase, se hace un `include Modulo`...
- ...y automágicamente se añaden los nuevos métodos

Cómo

- Se añaden métodos a un *módulo*, y luego se incluyen en las clases
- Los métodos se añaden como si fueran del *ejemplar*, no de la *clase* (no se pueden llamar directamente)
- En la clase, se hace un `include Modulo`...
- ...y automágicamente se añaden los nuevos métodos
- Se puede hacer para un *objeto* concreto, con el método `extend`

Ejemplo de Mixin

```
module Enumerable
  def collect
    # Algo con each
  end
  def grep
    # Algo con each
  end
end

class Array
  include Enumerable
  def each; ...; end
end

# Ya tenemos collect, grep, etc. en Array
```

Índice

Reflexión

- El dinamismo viene apoyado por la reflexión

Reflexión

- El dinamismo viene apoyado por la reflexión
- Las definiciones de clases, las crean o *añaden* métodos

Reflexión

- El dinamismo viene apoyado por la reflexión
- Las definiciones de clases, las crean o *añaden* métodos
- Las clases no son más que constantes, que podemos asignar, comparar...

Reflexión

- El dinamismo viene apoyado por la reflexión
- Las definiciones de clases, las crean o *añaden* métodos
- Las clases no son más que constantes, que podemos asignar, comparar...
- Podemos añadir métodos a cada *objeto*

Reflexión

- El dinamismo viene apoyado por la reflexión
- Las definiciones de clases, las crean o *añaden* métodos
- Las clases no son más que constantes, que podemos asignar, comparar...
- Podemos añadir métodos a cada *objeto*
- Podemos llamar a métodos y obtener constantes dinámicamente

Reflexión

- El dinamismo viene apoyado por la reflexión
- Las definiciones de clases, las crean o *añaden* métodos
- Las clases no son más que constantes, que podemos asignar, comparar...
- Podemos añadir métodos a cada *objeto*
- Podemos llamar a métodos y obtener constantes dinámicamente
- Podemos preguntar a qué métodos responde un objeto

Reflexión

- El dinamismo viene apoyado por la reflexión
- Las definiciones de clases, las crean o *añaden* métodos
- Las clases no son más que constantes, que podemos asignar, comparar...
- Podemos añadir métodos a cada *objeto*
- Podemos llamar a métodos y obtener constantes dinámicamente
- Podemos preguntar a qué métodos responde un objeto
- Podemos saber qué objetos (y clases, por tanto) existen en ese momento

Ejemplos de reflexión

```
class String
  def metodo_nuevo; puts "Soy nuevo"; end
  alias_method :old_strip, :strip
  def strip; puts "Tariro, tariro..."; old_strip; end
end
"".metodo_nuevo
"  con espacios, sin ropa     ".strip

def ClaseNueva; end
miClase = quieresString ? String : ClaseNueva
obj = miClase.new
```

Ejemplos de reflexión

```
class String
  def metodo_nuevo; puts "Soy nuevo"; end
  alias_method :old_strip, :strip
  def strip; puts "Tariro, tariro..."; old_strip; end
end
"".metodo_nuevo
"  con espacios, sin ropa     ".strip

def ClaseNueva; end
miClase = quieresString ? String : ClaseNueva
obj = miClase.new
```

Ejemplos de reflexión

```
class String
  def metodo_nuevo; puts "Soy nuevo"; end
  alias_method :old_strip, :strip
  def strip; puts "Tariro, tariro..."; old_strip; end
end
"".metodo_nuevo
"  con espacios, sin ropa     ".strip

def ClaseNueva; end
miClase = quieresString ? String : ClaseNueva
obj = miClase.new
```

Ejemplos de reflexión

```
class String
  def metodo_nuevo; puts "Soy nuevo"; end
  alias_method :old_strip, :strip
  def strip; puts "Tariro, tariro..."; old_strip; end
end
"".metodo_nuevo
"  con espacios, sin ropa     ".strip

def ClaseNueva; end
miClase = quieresString ? String : ClaseNueva
obj = miClase.new
```

Ejemplos de reflexión

```
class String
  def metodo_nuevo; puts "Soy nuevo"; end
  alias_method :old_strip, :strip
  def strip; puts "Tariro, tariro..."; old_strip; end
end
"".metodo_nuevo
"  con espacios, sin ropa     ".strip

def ClaseNueva; end
miClase = quieresString ? String : ClaseNueva
obj = miClase.new
```

Ejemplos de reflexión

```
class String
  def metodo_nuevo; puts "Soy nuevo"; end
  alias_method :old_strip, :strip
  def strip; puts "Tariro, tariro..."; old_strip; end
end
"".metodo_nuevo
"  con espacios, sin ropa     ".strip

def ClaseNueva; end
miClase = quieresString ? String : ClaseNueva
obj = miClase.new
```

Más ejemplos de reflexión

```
class <<obj
  def metodo_unico; puts "Solamente en obj"; end
end
obj.metodo_unico
"".metodo_unico    # NoMethodError

disponibles = App::Drivers.constants
controlador = App::Drivers.const_get('Pg').new
```

Más ejemplos de reflexión

```
class <<obj
  def metodo_unico; puts "Solamente en obj"; end
end
obj.metodo_unico
"".metodo_unico    # NoMethodError

disponibles = App::Drivers.constants
controlador = App::Drivers.const_get('Pg').new
```

Más ejemplos de reflexión

```
class <<obj
  def metodo_unico; puts "Solamente en obj"; end
end
obj.metodo_unico
"".metodo_unico    # NoMethodError

disponibles = App::Drivers.constants
controlador = App::Drivers.const_get('Pg').new
```

Más ejemplos de reflexión

```
class <<obj
  def metodo_unico; puts "Solamente en obj"; end
end
obj.metodo_unico
"".metodo_unico    # NoMethodError

disponibles = App::Drivers.constants
controlador = App::Drivers.const_get('Pg').new
```

Más ejemplos de reflexión

```
class <<obj
  def metodo_unico; puts "Solamente en obj"; end
end
obj.metodo_unico
"".metodo_unico    # NoMethodError

disponibles = App::Drivers.constants
controlador = App::Drivers.const_get('Pg').new
```

Más ejemplos de reflexión

```
class <<obj
  def metodo_unico; puts "Solamente en obj"; end
end
obj.metodo_unico
"".metodo_unico    # NoMethodError

disponibles = App::Drivers.constants
controlador = App::Drivers.const_get('Pg').new
```

Más ejemplos de reflexión

```
str = ""
str.methods.grep(/trip/)
metodo = :strip
str.send(metodo) if str.respond_to? metodo

ObjectSpace.each_object(Class) do |o|
  next unless o.name =~ /^TC_/
  puts "La clase #{o} empieza por TC_"
end
```

Más ejemplos de reflexión

```
str = ""
str.methods.grep(/trip/)
metodo = :strip
str.send(metodo) if str.respond_to? metodo

ObjectSpace.each_object(Class) do |o|
  next unless o.name =~ /^TC_/
  puts "La clase #{o} empieza por TC_"
end
```

Más ejemplos de reflexión

```
str = ""
str.methods.grep(/trip/)
metodo = :strip
str.send(metodo) if str.respond_to? metodo

ObjectSpace.each_object(Class) do |o|
  next unless o.name =~ /^TC_/
  puts "La clase #{o} empieza por TC_"
end
```

Más ejemplos de reflexión

```
str = ""
str.methods.grep(/trip/)
metodo = :strip
str.send(metodo) if str.respond_to? metodo

ObjectSpace.each_object(Class) do |o|
  next unless o.name =~ /^TC_/
  puts "La clase #{o} empieza por TC_"
end
```

Más ejemplos de reflexión

```
str = ""
str.methods.grep(/trip/)
metodo = :strip
str.send(metodo) if str.respond_to? metodo

ObjectSpace.each_object(Class) do |o|
  next unless o.name =~ /^TC_/
  puts "La clase #{o} empieza por TC_"
end
```

Más ejemplos de reflexión

```
str = ""
str.methods.grep(/trip/)
metodo = :strip
str.send(metodo) if str.respond_to? metodo

ObjectSpace.each_object(Class) do |o|
  next unless o.name =~ /^TC_/
  puts "La clase #{o} empieza por TC_"
end
```

¡Se acabó!

¿Qué más quieres?