Aprende a programar con Ruby

Andrés Suárez

Published with GitBook

Tabla de contenido

Este tutorial de Ruby está basado en The Ruby Study Notes, de Satish Talim, aunque he añadido algo de cosecha propia, y organizado las lecciones de otra manera. Además, para aquellos que sepan programar y quieran ver las capacidades de Ruby, he preparado este Ruby en 15 minutos, que es un resumen muy condensado de todo este tutorial. También lo recomiendo para aquellos que lo hayan terminado, y quieran repasar lo aprendido.

¿Qué es Ruby?

Ruby es un lenguaje multiplataforma, interpretado y orientado a objetos. Ruby fue diseñado por Yukihiro Matsumoto ('Matz') en 1993, con el Principio de la Menor Sorpresa.

> "Quería minimizar mi frustración mientras programo, y eso conllevaba minimizar mi esfuerzo. Este es el principal objetivo de Ruby. Quiero divertirme mientras programo. Después de lanzar Ruby y probarlo mucha gente, ellos me dijeron que sentían lo mismo que yo. Ellos fueron los que acuñaron el término de "Principio de Menor Sorpresa". Yukihiro Matsumoto

En el año 2004 hubo un boom en el interés por Ruby, debido a Ruby on Rails: el entorno para desarrollo web de David Heinemeier Hansson.

¿Cómo puede ayudarte?

En el libro de David Black "Ruby for Rails", él menciona que un buen conocimiento en Ruby puede ayudarte, como desarrollador de Rails, en cuatro formas:

1. Conocer mejor el código de tu aplicación (incluso el código que Rails escribe automáticamente).
2. Ser más productivo con Rails, incluso si dominas todas sus técnicas.
3. Familiarizarte mejor con el código de Rails, lo que te permitirá participar en discusiones sobre Rails y quizás poder ayudar detectando bugs o aportando patches.
4. Utilizar una poderosa herramienta para tareas de administración y organización conectadas con tu aplicación.

Algunas características

- **Libre de formato**: una cosa se puede hacer de distintas maneras. Escoge la que mejor se adapte a tu forma de trabajo.
- **Sensible a las mayúsculas**: dos palabras, aunque se diferencien solamente en una letra, por estar en mayúscula o minúscula, son dos cosas distintas. Por ejemplo, 'Dir' no es lo mismo que 'dir'.
- **Comentarios**: cualquier línea precedida por `#` es ignorada por el intérprete. Además, cualquier cosa que escribamos entre las líneas `=begin` y `=end` (empezando ambas en la primera columna de su correspondiente línea), también será ignorada.

```
# Comentario de una sola línea
```

```
=begin
Esto es
un comentario
de varias
líneas
=end
```

MUY IMPORTANTE: este último tipo de comentarios, no puede tener espacios a su izquierda, por que daría un error. Por lo tanto, si se quiere usar, siempre van pegados al margen izquierdo de la pantalla.

```
  =begin
```

```
  Este comentario multilínea
  da un error.
  =end
```

- **Delimitadores de instrucción**: varias instrucciones en una misma línea pueden ser separadas por un `;`, pero no son necesarios al final de una línea: este final de línea (o retorno de carro) se trata como un `;`. Si un final de línea acaba con un `\`, entonces el retorno de carro es ignorado, lo que permite tener una instrucción divida en varias líneas.

```
#Varias instrucciones en una misma línea
a =1; b=2; c=3
#es equivalente a:
a = 1
b = 2
c = 3
```

```
#Una instrucción en varias líneas
 a = 'enjuto mojamuto'
#es equivalente a:
a = "enjuto \
mojamuto"
```

- **Palabras clave**: también conocidas como palabras reservadas, son palabras en Ruby que no pueden ser usadas para otros propósitos, por ejemplo, como almacenar valores. Además, puedes estar acostumbrado a pensar que un valor 'falso' puede estar representado por 0 (en Ruby se evalúa a true), una cadena vacía o varias otras cosas.

- `false` y `nil`: En Ruby, todo esto es válido; de hecho, todo es cierto excepto las palabras reservadas `false` y `nil`.

Descargando Ruby

Como lenguaje multiplataforma, Ruby ha sido portado a distintos sistemas operativos y arquitecturas. Esto significa que si tu desarrollas un programa en un PC (por ejemplo), será posible ejecutarlo en otra máquina distinta como es un MAC (por poner otro ejemplo).

Las siguientes instrucciones son para instalar Ruby en un PC. Para otras plataformas, ver el primer capítulo de la *Why's (poignante) guide to ruby*.

La forma más sencilla de instalar Ruby en un PC es mediante el Ruby One-Click Installer. Después de descargarlo instálalo aceptando todo por defecto. Al instalarse, las Variables de Entorno del Sistema son actualizadas, de tal forma que se incluya el directorio del ejecutable de Ruby: gracias a esto, podrás ejecutarle desde cualquier drectorio en tu PC.

La instalación de Ruby incluye la primera edición del libro "Programming Ruby" y el editor SciTe.

Directorios de la instalación

Supongamos que la instalación de Ruby fue en c:/ruby. Esta instalación creó una serie de directorios:

- `c:/ruby/bin` es donde los ejecutables son instalados (incluyendo ruby e irb).
- `c:/ruby/lib/ruby/1.8` aquí están programas escritos en ruby. Estos ficheros son librerías de Ruby: proveen funcionalidades que puedes incorporar en tus programas.
- `c:/ruby/lib/ruby/1.8/i386-mswin32` contiene una serie de extensiones específicas del PC. Los ficheros en esta terminación acaban en .so o .dll (dependiendo de la plataforma). Estos ficheros con extensiones programadas en lenguaje C; dicho de otra forma: son fihceros binarios, compilados durante el proceso de instalación, que se pueden ejecutar desde Ruby.
- `c:/ruby/lib/ruby/site_ruby` aquí es donde el administrador de tu sistema y/o tú podéis almacenar las extensiones y librerías de terceras partes: código escrito por ti mismo, o por otros.
- `c:/ruby/lib/ruby/gems` es el sistema Ruby-Gemes, encargado de la instalación de nuevas herramientas.
- `c:/ruby/src` es donde se halla el código fuente de Ruby.
- `c:/ruby/samples/RubySrc-1.8.6/sample` aquí podrás encontrar algunos programas ejemplo escritos en Ruby.

El primer programa

Usemos el editor SciTE: Start/Programas/Ruby/SciTe. Se abre el editor. Para cambiar los parámetros de arranque, Options/Opens Global y allí modificar:

- tabsize=2, indent.size=2 (tamaño del tabulador, y el identado) position.width=-1, position.height=-1 (arrancar maximizado)

Una vez cambiados los valores, pulsar `Ctrl+S` y `Ctrl+S`. Para modificar estos dos valores, otra forma es:

- `Ctrl+Shift+I` - abre un diálogo donde modificar los valores anteriores. Tienen que ser =2.
- `F11` - para maximizar minimizar la ventana.

Lo último que falta antes escribir los programas, es un abrir una ventana de terminal para ver los resultados de los programas. Para ello hay que pulsar F8. Una vez ajustado el SciTE, lo siguiente es crear un directorio donde ir guardando los programas que vayamos creando.

'Hola'

En la ventana izquierda de SciTE (tiene que haber 2 ventanas después de pulsar F8), escribir:

```
#p001hola.rb
puts 'Hola'
```

Ahora hay que guardar y ejecutar el programa. Las normas dicen que el nombre del fichero o directorio, es el nombre en minúsculas de la clase/módulo (más adelante se hablará de clases y módulos). Por ejemplo, la clase Foo está en el fichero foo.rb. Para guardar el fichero: File\Save As…Sálvalo como `p001hola.rb`. Todos los ficheros de código Ruby tienen la terminación "rb". Ahora ejecuta el programa: pulsa F5. En la ventana de output aparecerá la palabra "Hola".

En Ruby, la ejecución de las instrucciones del programa, va de la primera a la última:

```
#p001hola.rb
```

No hace nada. Todas las líneas precedidas por `#` son comentarios. Y los comentarios se ignoran al ejecutarse el programa.

```
puts 'Hello'
```

`puts` significa "poner string". String es una cadena de texto. Esta instrucción saca por el output cualquier cosa que pongamos a su derecha.

Normas de código

Los paréntesis son opcionales cuando usamos un método, en este caso puts. Las siguientes instrucciones son correctas:

```
foobar
foobar()
foobar(a, b, c)
foobar a, b, c
```

En Ruby, todo desde un número entero a una cadena de texto, es un objeto. Hablaremos más de esto. Y cada objeto tiene sus propios métodos (o instrucciones o funciones) que pueden ser usados para hacer cosas muy útiles. Para usar un método, necesitas poner un '.' después del nombre del objeto, y luego poner el nombre del método. Algunos métodos como puts y gets no necesitan estar asociados a un objeto, y por tanto pueden ser usados desde cualquier parte. Cuando se ejecuta una aplicación en Ruby, siempre se crea un objeto llamado main de la clase Object: este objeto es el que tiene dentro los métodos Kernel. De todo esto se hablará más adelante.

Todo esto se puede verificar por el siguiente programa (no te preocupes si no entiendes el programa, más adelante se explicará):

```
puts 'Soy una clases = ' + self.class.to_s
puts 'Soy un objeto = ' + self.to_s
print 'Los métodos del objeto son = '
puts self.private_methods.sort
```

Observaciones

- Programadores de C y Java - no se necesita escribir un método main.
- Los strings son secuencias de caracteres entre simple o dobles comillas. Las comillas simples son más eficientes que las dobles. Se explicará más adelante.
- Ruby es un lenguaje interpretado, entonces no hace falta recompilar para ejecutar un programa.
- Las normas de código en Ruby, establecen que el nombre del fichero/directorio tiene que ser el nombre de la clase/módulo en minúsculas, añadiendo la extensión .rb

Números

En Ruby, los números sin la coma decimal son llamados enteros, y los enteros con decimales son llamados coma-flotantes, o más sencillamente, flotantes.

```
puts 1 + 2
puts 10 - 11
puts 2 * 3
#División: cuando divides dos enteros, obtienes un entero:
puts 3 / 2
#si quieres obtener el resultado de decimal,
#al menos uno de los dos tiene que ser decimal
puts 3.0 / 2
puts 3 / 2.0
puts 1.5 / 2.6
```

Los números en Ruby son objetos de la clase Fixnum o Bignum: estas clases representan números enteros de distintos tamaños. Ambas clases descienden de la clase Integer (en inglés, integer=entero). Los números coma-flotantes son objetos de la clase Float (en inglés, float=flotante).

Veamos el ejemplo que Peter Cooper nos propone, sacado de su libro "Beginning Ruby" (no importa que todavía no seas capaz de entender todo el código):

```
=begin
Problema del tablero de ajedrez:
si en la primera casilla ponemos un grano,
y duplicamos la cantidad de granos en la siguiente,
y así hasta rellenar el tablero,
¿cuántos granos tendremos?
=end

granos = 1
64.times do |escaque|
puts "En el escaque #{escaque+1} hay #{granos}"
granos *= 2
end
```

Al final tenemos 2.2.2...2.2=2^{64} granos en la última casilla…¡trillones de granos! Esto demuestra que Ruby es capaz de manejar números extremadamente grandes, y a diferencia de otros lenguajes de programación, no hay límites en esos números. Ruby hace esto gracias a las distintas clases antes mencionadas:

- **Fixnum**: maneja los números pequeños
- **Bignum**: maneja los números grandes (en inglés, big=grande). Ruby escogerá cuál usar, y tú únicamente tendrás que preocuparte por lo que quieras hacer con ellos.

Operadores y precedencias

Echémosle un ojo a los operadores de Ruby (Half Fulton - The Ruby Way). Están ordenados de mayor a menor rango de precendencia; dicho de otra forma, los de la parte superior de la tabla, son los primeros en ejecutarse.

operador	significado
:	Alcance (scope)
[]	Índices

**	Exponentes
+ - ! ~	Unarios: pos/neg, no,…
* / %	Multiplicación, División,…
+ -	Suma, Resta,…
« »	Desplazadores binarios,…
&	'y' binario
, ^	'or' y 'xor' binarios
> >= < <=	Comparadores
== === <=> != =~ !~	Igualdad, inegualdad,…
&&	'y' booleano
||	'o' booleano
.. …	Operadores de rango
= (+=, -=,...)	Asignadores
?:	Decisión ternaria
not	'no' booleano
and, or	'y', 'o' booleano

Destacar que:

Los paréntesis funcionan de la misma forma que en las matemáticas: cualquier cosa dentro de ellos es calculado en primer lugar. O dicho más técnicamente: tienen más precedencia. Los operadores incremento y decremento (`++` y `--`) no están disponibles en Ruby, ni en su forma "pre" ni en su forma "post".

El operador módulo

El operador módulo, que nos da el resto de una división, se comporta como sigue:

```
puts (5 % 3)     # imprime  2
puts (-5 % 3)    # imprime  1
puts (5 % -3)    # imprime -1
puts (-5 % -3)   # imprime -2
```

Ejercicio

Escribir un programa que diga cuantos minutos hay en un año de 365 días.

Strings

Los strings (o cadenas de texto) son secuencias de caracteres entre comillas simples o comillas dobles. `''` (dos comillas simples) no tienen nada: podemos llamarlo string vacío.

```
puts "Hola mundo"
# Se puede usar " o ' para los strings, pero ' es más eficiente.
puts 'Hola mundo'
# Juntando cadenas
puts 'Me gusta' + ' Ruby'
# Secuencia de escape
puts 'Ruby\'s party'
# Repetición de strings
puts 'Hola' * 3
# Definiendo una constante
# Más adelante se hablará de constantes
PI = 3.1416
puts PI
```

En Ruby las cadenas son mutables: se pueden modificar. Ruby almacena las cadenas como secuencias de bytes.

El acento grave `

Hay unos strings especiales que se diferencian por usar como delimitador el acento grave `:

```
puts `dir`
```

El string entre los acentos, es enviado al sistema operativo como un comando a ser ejecutado. El resultado devuelto por el sistema, es recogido por Ruby.

Interpolación

Con la interpolación nos referimos al proceso de insertar el resultado de una expresión dentro de un string. La forma de hacerlo es mediante `#{ expresión }`. Ejemplo:

```
puts "100 * 5 = #{100 * 5}"
```

La sección `#{100*5}` se ejecuta y pone el resultado en esa posición. Es decir, 500.

Comillas (') vs comillas dobles (")

La diferencia entre ambas formas, es el tiempo que se toma Ruby en cada una: mientras que con las simples comillas, Ruby hace muy poco; en las dobles comillas, Ruby tiene que hacer más trabajo:

1. busca posibles sustituciones: las secuencias de escape (las que empiecen por un) son sustituidas por su valor binario.
2. busca posibles interpolaciones: en las secuencias con #{expresión}, se calcula la expresión, y se sustituye el bloque entero por su resultado.

```ruby
def di_adios(nombre)
  resultado = "Buenas noches, #{nombre}"
  return resultado
end

puts di_adios('Pepe')

=begin
  Como los métodos devuelven el valor
  de la última línea, no hace falta
  el return.
=end

def di_adios2(nombre)
  resultado = 'Buenas noches, ' + nombre
end
puts di_adios2('Pepe')

=begin
  Ahora, en vez de usar ", usamos ',
  utilizando la concatenación de strings
  para obtener el mismo resultado.
=end
```

String#length

String#length devuelve el número de bytes de una cadena.

```ruby
string = "Esto es una cadena"
string.length # => 18
```

`string.length` devuelve 18. Cuenta todas las letras, incluidos los espacios en blanco.

Variables

Para almacenar un número o un string en la memoria del ordenador, con el fin de usarlos en cálculos posteriores, necesitas dar un nombre a ese número o string. En programación este proceso es conocido como **asignación**.

```
#Ejemplos de asignaciones

s = 'Hello World!'
x = 10
```

Las variables locales en ruby son palabras que deben:

1. empezar con un letra minúscula o un guión bajo `_`.
2. estar formadas por letras, números y/o guiones bajos.

Cuando Ruby encuentra una palabra, la interpreta como: una variable local, un método o una palabra clave. Las **palabras claves** no pueden ser usados como variables. Por ejemplo `def` es una palabra clave: sólo se puede usar para definir un método. `if` también es una palabra clave: gran parte del código consta de instrucciones condicionales que empiezan con `if`, por eso sería muy confuso si pudiese usarse como variable.

Los métodos pueden ser palabras, como `start_here`, `puts` o `print`. Cuando Ruby encuentra una palabra decide qué es de la siguiente forma:

1. si hay un signo de igualdad (=) a la derecha de la palabra, es una variable local a la que se le asigna un valor.
2. si la palabra es una palabra clave, entonces es una palabra clave. Ruby tiene una lista interna para poder reconocerlas.
3. si no se cumple ninguno de los anteriores casos, Ruby asume que es un método.

```
# Definición de una constante
PI = 3.1416
puts PI
# Definición de una variable local
myString = 'Yo amo mi ciudad, Vigo'
puts myString

=begin
Conversiones
to_i - convierte a número entero
to_f - convierte a número decimal
to_s - convierte a string
=end

var1 = 5
var2 = '2' #fijarse que es un texto
puts var1 + var2.to_i

=begin
<< marca el comienzo de un string
    y es seguido de ' o ''. Aquí añadimos
    el string junto con el retorno de carro (\n).
=end

a = 'molo '
a<<'mucho.
Molo mazo...'
puts a

=begin
    ' o " son los delimitadores de un string.
    En este caso, podemos sustituirlos por END_STR.
    END_STR es una constante delimitador de strings.
```

```
=end

a = <<END_STR
This is the string
And a second line
END_STR
puts a
```

En el ejemplo, `var2.to_i` el punto significa que el método `to_i` es enviado a la variable `var2`, que este caso es un string: transformamos el string en un número para poder sumarlos. Cuando hablemos de objetos, veremos que se puede decir que la `var2` es el receptor de `to_i`. Por lo tanto, cuando aparezca un punto en una posición inexplicable, habrá que interpretarlo como un método (la parte derecha) que es enviado a un objeto (la parte izquierda).

Interpretación dinámica

Por interpretación dinámica, se entiende que no hace falta especificar qué tipo de variable se va a manejar: si parece un número, problablemente sea un número; si parece una cadena, probablemente lo sea. El método class devuelve el tipo de clase de un objeto:

```
s = 'hello'
s.class # String
```

Otro ejemplo:

```
# Ruby es dinámico
x = 7        #número entero
x = "house"  #string
x = 7.5      #número real
```

Alcance

El alcance es una propiedad de las variables: se refiere a su visibilidad (aquella región del programaa donde la variable puede utilizarse). Los distintos tipos de variables, tienen distintas reglas de alcance. Hablemos de dos tipos de variables: las globales y las locales.

Alcance global y variables globales

Empezarmos con el alcance que menos se usa, pero no por ello menos importante: un alcance global significa que alcanza a todo el programa. Desde cualquier parte del programa, puede usarse la variable. Las variables globales son las que tienen alcance global.

Las variables globales se distinguen porque están precedidas del signo dólar `$`. Pueden ser vistas desde cualquier parte del programa, y por tanto pueden ser usadas en cualquier parte: nunca quedan fuera de alcance. Sin embargo, las variables globales son usadas muy poco por los programadores experimentados.

Variables globales por defecto

El intérprete Ruby tiene por defecto un gran número de variables globales iniciadas desde el principio. Son variables que almacenan información útil que puede ser usada en cualquier momento y parte del programa.

Por ejemplo, la variable global `$0` contiene el nombre del fichero que Ruby está ejecutando. La variable global `$:` contiene los directorios en los que Ruby busca cuando se carga un fichero que no existe en el directorio de trabajo. `$$` contiene el id (identidad) del proceso que Ruby está ejecutando. Y hay muchas más.

Alcance local

Nota: no te preocupes si no entiendes esto ahora.

Se puede intuir mirando el código dónde empieza y acaba el alcance de las variables locales, basándonos en:

El nivel más alto (fuera de todos los bloques de definición) tienen su propio alcance. Cada bloque de definición de una clase o módulo tienen su propio alcance, incluso los bloques anidados. Toda definición de un método (def) tiene su propio alcance.

Variables

En Ruby, todo lo que se manipula es un objeto, y el resultado de esas operaciones también son objetos. La única forma que tenemos de manipular los objetos, son los **métodos**:

```
5.times { puts "Ratón!\n" } #se hablará más tarde de bloques
"A los elefantes le gustan los cacahuetes".length
```

Si los objetos (como los strings, números,…) son los nombres, entonces los métodos son los verbos. Todo método necesita un objeto. Es fácil decir qué objeto recibe el método: el que está a la izquierda del punto. Algunas veces, puede que no sea obvio. Por ejemplo, cuando se usa `puts` y `gets`, ¿dónde están sus objetos? Nada más iniciarse el intérprete, estamos dentro de un objeto: el objeto `main`. Por tanto, al usar puts y gets, estamos mandando el mensaje al objeto `main`.

¿Cómo podemos saber dentro de qué objeto estamos? Usando la variable `self`.

```
puts self
```

Escribiendo métodos

Un bloque de instrucciones que define un método, empieza por la palabra def y acaba por la end. Los parámetros son la lista de variables que van entre paréntesis. Aunque en Ruby, dichos paréntesis son opcionales: `puts`, `p` y `gets` son muy usados, y por ello que el uso de paréntesis sea opcional. En Rails, se llama a los métodos sin paréntesis.

Un método devuelve el valor de su última línea. Por norma, es recomendable dejar una línea en blanco entre las definiciones de métodos:

```
#metodos.rb

# Definición de un método
def hello
  puts 'Hola'
 end
 #uso del método
 hello

 # Método con un argumento
def hello1(nombre)
  puts 'Hola ' + nombre
  return 'correcto'
end
puts(hello1('Pedro'))

# Método con un argumento (sin paréntesis, no funciona en versiones nuevas)
def hello2 nombre2
  puts 'Hola ' + nombre2
  return 'correcto'
end
puts(hello2 'Juan')
```

Esto es lo que obtenemos

```
>ruby metodos.rb
Hola
Hola Pedro
```

```
correcto
Hola Juan
correcto
metodos.rb:18 warning: parenthesize argument(s) for future version
>Exit code: 0
```

Los métodos bang (!)

Los métodos que acaban con una `!` son métodos que modifican al objeto. Por lo tanto, estos métodos son considerados como peligrosos, y existen métodos iguales, pero sin el `!`. Por su peligrosidad, el nombre "bang". Ejemplo:

```
a = "En una lugar de la mancha"

#método sin bang: el objeto no se modifica
b = a.upcase
puts b
puts a

#método con bang: el objeto se modifica
c = a.upcase!
puts c
puts a
```

Normalmente, por cada método con `!`, existe el mismo método sin `!`. Aquellos sin bang, nos dan el mismo resultado, pero sin modificar el objeto (en este caso el string). Las versiones con `!`, como se dijo, hacen la misma acción, pero en lugar de crear un nuevo objeto, transforman el objeto original.

Ejemplos de esto son: `upcase` / `upcase!`, `chomp` / `chomp!`,...En cada caso, si haces uso de la versión sin `!`, tienes un nuevo objeto. Si llamas el método con `!`, haces los cambios en el mismo objeto al que mandaste el mensaje.

Alias

```
alias nuevo_nombre nombre_original
```

alias crea un nuevo nombre que se refiere a un método existente. Cuando a un método se le pone un alias, el nuevo nombre se refiere al método original: si el método se cambia, el nuevo nombre seguirá invocando el original.

```
def viejo_metodo
   "viejo metodo"
end
alias nuevo_metodo viejo_metodo
def viejo_metodo
  "viejo metodo mejorado"
end
puts viejo_metodo
puts nuevo_metodo
```

En el resultado, vemos como `nuevo_metodo` hace referencia al `viejo_metodo` sin modficar:

```
viejo metodo mejorado
viejo metodo
```

Métodos perdidos

Cuando mandas un mensaje a un objeto, el objeto busca en su lista de métodos, y ejecuta el primer método con el mismo nombre del mensaje que encuentre. Si no encuetra dicho método, lanza una error `NoMethodError`.

Una forma de solucionar esto, es mediante el método `method_missing`: si definimos dicho método dentro de una clase, se ejecuta este método por defecto:

```
class Dummy
  def method_missing(m, *args)
    puts "No existe un metodo llamado #{m}"
  end
end
```

`Dummy.new.cualquier_cosa` obtenemos:

```
No existe un metodo llamado cualquier_cosa
```

Por lo tanto, `method_missing` es como una red de seguridad: te da una forma de manejar aquellos métodos que de otra forma darían un error en tu programa.

Argumentos

Valores por defecto

Ruby deja especificar los valores por defecto de los argumentos, que son usados si no se especifica un valor explícitamente. Se hace esto mediante el operador de asignación `=`:

```
#argumentos.rb

def mtd(arg1="Dibya", arg2="Shashank", arg3="Shashank")
   "#{arg1}, #{arg2}, #{arg3}."
end
puts mtd
puts mtd("ruby")
```

Hemos usado el operador interpolación `#{ }`: se calcula la expresión entre paréntesis, y el resultado se añade al string. Lo que obtenemos es:

```
>ruby argumentos.rb
Dibya, Shashank, Shashank.
ruby, Shashank, Shashank.
>Exit code: 0
```

Número de argumentos variable

Ruby permite escribir funciones que acepten un número variable de argumentos. Por ejemplo:

```
def foo(*mi_string)
  mi_string.each do |palabras|
    puts palabras
```

```
  end
end

foo('hola', 'mundo')
foo()
```

El **asterisco** indica que el número de argumentos puede ser el que se quiera. En este ejemplo, el asterisco toma los argumentos y los asigna a un array (o vector de elementos) llamado `mi_string`. Haciendo uso de ese asterisco, incluso se pueden pasar cero argumentos; que es lo que pasa con `foo()`.

No hay máximo número de argumentos que podamos pasar a un método.

Argumentos opcionales

Si se quieren incluir argumentos opcionales, tienen que venir después de los argumentos no opcionales:

```
def arg_opc(a,b,*x) # bien
def arg_opc(a,*x,b) # mal
```

Los argumentos se interpretan de izquierda a derecha, por eso es importante que los argumentos no opcionales vayan en primer lugar. Si los pusiésemos en último lugar, no sabríamos decir donde acaban los argumentos opcionales y donde empiezan los no opcionales.

```
=begin
Ejemplo de como los argumentos se
interpretan de izquierda a derecha
=end

def mtd(a=99, b=a+1)
  [a,b]
end
puts mtd
```

Introduciendo datos (gets)

Lecciones atrás vimos el método `puts` que saca datos por la pantalla. ¿Cómo podemos introducir nuestros propios datos? Para esto gets (get=coger, s=string) y chomp son de ayuda. Veamos el siguiente ejemplo:

```
# gets y chomp
puts "¿En qué ciudad te gustaría vivir?"
STDOUT.flush
ciudad = gets.chomp
puts "La ciudad es " + ciudad
```

El ejemplo superior, al ser ejecutado en SciTe, clickea en la pantalla de output y pon el nombre de tu ciudad favorita. `STDOUT` es una constante global que almacena las salidas del programa. flush vacía cualquier dato almacenado, y por lo tanto, limpiará cualquier resultado anterior. `chomp` es un método para strings y `gets` almacena strings que provienen del teclado. El problema es que gets almacena lo escrito y el caráter `\n` (retorno de carro); `chomp` lo que hace es borrar el carácter: `\n`.

RAILS: los datos vienen de muchos sitios. En la típica aplicación de Rails, vienen de una base de datos. Como un desarrollador de Rails, puedes usar con frecuencia algunos de estos métodos, porque Rails recoge los datos que los usuarios escriben en los formularios Web.

Ejercicio

Escribe un programa que pregunte por la temperatura en grados Fahrenheit. El programa usará este dato, y hallará el equivalente en grados Celsius. El resultado final lo mostrará en pantalla con dos decimales. (Celsius (°C) = [Fahrenheit (°F) - 32] / 1.8)

Nota: para formatear un resultado a dos decimales, hay dos opciones:

1. Usar el método `format`. Por ejemplo:

```
x = 45.5678
puts format("%.2f", x)
```

1. Otra forma es la función `round`:

```
puts (x*100).round/100.0
```

Normas en los nombres

Un nombre es una letra mayúscula, una letra minúscula o un guión bajo, seguido por cualquier combinación de mayúsculas, minúsculas, números o guiones bajos.

Los nombres en Ruby se usan para referirse a constantes, variables, métodos, clases y módulos. La primera letra de un nombre ayuda a Ruby a distinguirlos. Algunos nombres, son palabras reservadas y no pueden usarse como variable, método, clase o módulo. El conjunto de las minúsculas abarca de la `a` a la `z` incluyendo el guión bajo `_`. El conjunto de las mayúsculas abarca de la `A` a la `Z` y los números (del `0` al `9`).

Variables

Las variables contienen cualquier tipo de dato. El propio nombre de la variable, muestra su alcance (local, global,…):

- Una variable local consiste en una letra minúscula o guión bajo seguido de cualquier mayúscula o minúscula. P.ej.: sunil, _z, rock_and_roll
- Una variable de un objeto (más adelante se hablará de clases y objetos) empieza con `@`, seguido de cualquier mayúscula o minúscula.

```
@sign
@_
@Counter
```

- Una variable de clase empieza con `@@` seguido por cualquier mayúscula o minúscula.

```
@@signo
@@_
@@Counter
```

- Una variable global empieza por `$`, seguido por cualquier carácter(no sólo mayúsculas o minúsculas).

```
$counter
$COUNTER
$-x.
```

Constantes

Una constante empieza por una letra mayúscula, seguido por cualquier mayúscula o minúscula. Los nombres de clases y de módulos son constantes, y siguen unas normas.

```
module MyMath
PI=3.1416
class Perro
```

Los nombres de métodos deben empezar por una minúscula (letra o guión bajo). La `?` y la `!` son los únicos caracteres ajenos al grupos de las mayúsculas y minúsculas, que se permiten como sufijos de los métodos. Más adelante se explicará su uso.

Por norma, se usa el guión bajo para separar palabras compuestas en los nombres de métodos y de variables. Para los nombres de clases, módulos y constantes, la norma dice de usar letras mayúsculas en vez de guiones bajos, para

distinguirlas. Ejemplos:

- variables y métodos

```
real_madrid
futbol_club_barcelona
```

- clases, módulos y constantes:

```
RealMadrid
FutbolClubBarcelona
```

Hay que notar que una variable puede referirse a distintos valores a lo largo del tiempo. Una constante en Ruby puede ser una referencia a un objeto. Las constantes son creadas en el momento de su primera asignación, normalmente en la definición de una clase o un módulo; no deben estar definidas en los métodos. Se puede variar el valor de una constante, pero esto genera un valor de aviso.

Rangos

El principal uso y quizás el más apropiado para los rangos, es expresar una secuencia: las secuencias tienen un punto inicial y un punto final, y una forma de producir los sucesivos valores entre ambos. En Ruby, esas secuencias son creadas usando los operandos `..` y `…`

`..` genera una secuencia donde los puntos límites están incluidos.

```
(1..3).to_a

#es la secuencia 1, 2, 3
```

`…` genera una secuencia en la que no está incluida el límite superior.

```
(1...5).to_a

#equivale a 1, 2, 3, 4
```

En Ruby los rangos no son almacenados como una lista: los rangos se almacenan como un objeto Range, y contiene referencias a dos objetos Fixnum (su límite superior e inferior). Se puede convertir un rango en un array (array = lista, conjunto ordenado de elementos), mediante el método `to_a`.

```
(1..10).to_a

#obtenemos [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Métodos de rangos

Los rangos en ruby tienen diversos métodos:

```
nums = -1...9
puts nums.include?(5)   # true
puts nums.min           # -1
puts nums.max           # 8
puts nums.reject {|i| i < 5} # [5, 6, 7, 8]
```

Uno de los usos útiles de los rangos, es comprobar si un determinado valor está en el intervalo representado por el rango. Para eso usamos el operador `===` :

```
(1..10) === 5       # true
(1..10) === 15      # false
(1..10) === 3.14159 # true
('a'..'j') === 'c'  # true
```

Arrays

Un array (o lista) es un conjunto ordenado: cada posición en la lista es una variable que podemos leer y/o escribir.

```
# Arrays (o vectores)

# array vacío
vec1 = []

# Los índices empiezan desde el cero (0,1,2,...)
nombres = ['Satish', 'Talim', 'Ruby', 'Java']
puts nombres[0]
puts nombres[1]
puts nombres[2]
puts nombres[3]
# si el elemento no existe, se devuelve nil
puts nombres[4]
# pero podemos añadir a posteriori más elementos
nombres[3] = 'Python'
nombres[4] = 'Rails'
```

Un array puede ser un conjunto de elementos distintos:

```
=begin
  un array cuyos elementos
  apuntan a otros tres objetos:
  un decimal, un string y un array
=end
sabor = 'mango'
vec4 = [80.5, sabor, [true, false]]
puts vec4[2]
```

Usando %w

Algunas veces, crear arrays de palabras puede ser tedioso debido a tantas comillas y comas. Afortunadamente, Ruby tiene una forma más cómoda para hacerlo:

```
nombres1 = [ 'ann', 'richard', 'william', 'susan', 'pat' ]
puts nombres1[0] # ann
puts nombres1[3] # susan

# esto es más sencillo y más rápido:
nombres2 = %w{ ann richard william susan pat }
puts nombres2[0] # ann
puts nombres2[3] # susan
```

El método each

El método each extrae cada elemento del array dentro de la variable que se le especifique (que irá entra dos barras | |), que se usará en bloque do…end.

```
ciudades = %w{ Pune Mumbai Bangalore }
ciudades.each do |ciudad|
  puts '¡Me gusta ' + ciudad + '!'
  puts '¿A ti no?'
```

```
end

# El método delete borra un elemento
ciudades.delete('Mumbai')
ciudades.each do |ciudad|
  puts '¡Me gustaba '+ciudad+'!'
  puts '¿A ti ya no?'
end
```

Por lo tanto el método each nos permite hacer una cosa (la que sea) con cada objeto del array. En el ejemplo, fuimos elemento por elemento del array sin usar los índices. Hay que destacar:

- Los variable entre los "postes" se refiere a cada ítem del array a medida que avanzamos en el loop. Se puede usar cualquier nombre, pero es mejor dotarlo de cierto significado.
- El `do…end` identifica el bloque de código que se ejecutará con cada elemento del array. Los bloques son usados intensivamente en Ruby, y se tratarán en profundidad más adelante. Es posibles sustiturlos por las llaves de inicio y fin.

Otros métodos

```
vec = [34, 12, 1, 38]
puts vec.sort
puts vec.length
puts vec.first
puts vec.last
```

Obteniendo arrays

Un método puede devolver un array:

```
def num_cuadrado(num)
  cuadrado = num * num
  return num, cuadrado
end

=begin
  el método nos devuelve
  un array con el número
  y su cuadrado
=end

x=3
num_cuadrado(x)

=begin
  si queremos almacenar el resultado
  hay que hacerlo por
  asignación en paralelo
=end

num, cuadrado = num_cuadrado(x)
```

Ejercicios

- Escribe un programa tal que, dado un array numérico, calcule la suma de sus elementos. Por ejemplo, `array = [1, 2, 3, 4, 5]`
- Escribe un programa tal que, dado un array de números, diga de cada uno si es par o impar. Por ejemplo, `array =`

```
[12, 23, 456, 123, 4579]
```

Bloques

Un bloque es una porción de código encerrada entre paréntesis `{}` o entre `do…end`. Por lo tanto, un bloque es una forma de agrupar instrucciones, y solo puede aparecer después de usar un método: el bloque empieza en la misma línea que usa el método. El código dentro del bloque no es ejectuado en el instante que el intérprete de Ruby lo encuentra: Ruby se recordará del bloque (variables locales, …) y después entra en el método, ejecutando el bloque cuando es preciso.

Supongamos que existen dos métodos llamados greet1 y greet2:

```
#greet1, no necesita argumentos
greet1 {puts 'Hola'}

#greet2, necesita un argumento
greet2 ("argumento_cualquiera") {puts 'Hola'}
```

Lo usual es usar las `{}` para bloques de una línea y el `do…end` para más de una línea.

El método yield

Un método puede usar el bloque mediante la palabra yield:

```
def metodo
  puts 'Comienzo del metodo'
  yield
  yield
  puts 'Final del metodo'
end

metodo{puts 'Dentro del bloque'}
```

La salida es:

```
'Comienzo del metodo'
'Dentro del bloque'     # primer yield
'Dentro del bloque'     # segundo yield
'Final del metodo'
```

Lo que sucede es que en el momento que el intérprete llega al yield, se ejecuta el código dentro del bloque, y luego se retorna al método.

Argumentos en los bloques

En los bloques se pueden usar argumentos especificándolos dentro de dos barras verticales `| |`. Y si se usan, en el yield no podemos olvidar darles valor:

```
def metodo
  yield('hola', 99)
end

metodo{|str,num| puts str + ' ' + num.to_s}  #hola 99
```

Un bloque de código devuelve un valor: el valor de la última expresión evaluada. Y este valor devuelto por yield, puede usarse dentro del método que invoca el bloque.

Los procs

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

```
prc = lambda{ "hola" }
```

Un bloque creado con lambda actúa como un método: si no especificas el número correcto de argumentos, no puedes llamar al bloque. La clase Proc tiene un método para llamar al bloque: el método `call`.

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

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

La salida es:

```
Hola
Gracias
```

Para usar argumentos con lambda:

```
aBlock = lambda { |x| puts x }
aBlock.call 'Hola Mundo!'
```

La salida es:

```
Hola Mundo!
```

Los procs son muy útiles por que:

- No puedes pasar métodos dentro de otros métodos (usarlos como argumentos); pero si puedes usar procs como argumentos.
- Los métodos no pueden devolver otros métodos; pero sí pueden devolver un procs.

```
#uso de procs como argumentos

def metod1 proc1
  puts 'Principio del metodo'
  proc1.call
  puts 'Final del metodo'
end

hola = lambda do
  puts 'Hola'
```

```
end

metod1 hola
```

La salida es:

```
Principio del metodo
Hola
Final del metodo
```

Expresiones Regulares

Las expresiones regulares, aunque crípticas, son una poderosa herramienta para trabajar con texto. Son usadas para reconocer patrones y procesar texto. Una expresión regular es una forma de especificar un patrón de caracteres, que será buscado en un string. En Ruby, se crean las expresiones regulares entre `//`. Son objetos del tipo Regexp y pueden ser manipuladas como tales.

```
//.class #  Regexp
```

La forma más simple de encontrar si una expresión (también funciona con strings) está dentro de un string, es usar el método match o el operador `=~`:

```
m1 = /Ruby/.match("El futuro es Ruby")
puts m1         # "Ruby", puesto que encontró la palabra
puts m1.class   # devuelve MatchData; devuelve "nil" si no se encuentra

# operador =~:
m2 = "El futuro es Ruby" =~ /Ruby/
puts m2 # 13 -> posición donde empieza la palabra "Ruby"
```

Construyendo expresiones regulares

Cualquier carácter que vaya entre barras, se busca exactamente:

```
/a/ # se busca la letra a, y cualquier palabra que la contenga
```

Algunos caracteres tienen un significado especial en las expresiones regulares. Para evitar que se procesen, y poder buscarlos, se usa la secuencia de escape `\`.

```
/\?/
```

La `\` significa "no trates el siguiente carácter como especial". Los caracteres especiales incluyen: `^`, `$`, `?`, `.`, `/`, `\`, `[`, `]`, `{`, `}`, `(`, `)`, `+` y `*`.

El comodín . (punto)

Algunas veces, se busca cualquier caracter en una posición determinada. Esto se logra gracias al `.`. Un punto, busca cualquier carácter, excepto el retorno de carro.

```
/.azado/
```

Busca `mazado` y `cazado`. También encuentra `%azado` y `8azado`. Por eso hay que tener cuidado al usar el punto: puede dar más resultados que los deseados. Sin embargo, se pueden poner restricciones a los resultados, especificando las clases de caracteres buscadas.

Clases de caracteres

Una clase de carácter es una lista explícita de caracteres. Para ello se usan los corchetes:

```
/[mc]azado/
```

De esta forma, especificamos la búsqueda de `azado` precedido por `c` o `m`: solamente buscamos `mazado` o `cazado`.

Dentro de los corchetes, se puede especificar un rango de búsqueda.

```
/[a-z]/ # encuentra cualquier minúscula
/[A-Fa-f0-9]/ # encuentra cualquier número hexadecimal
```

Algunas veces se necesita encontrar cualquier carácter menos aquellos de una lista específica. Este tipo de búsqueda se realiza negando, usando `^` al principio de la clase.

```
/[^A-Fa-f0-9]/ # encuentra cualquier carácter, menos los hexadecimales
```

Algunos caracteres son tan válidos, que tienen su abreviación.

Abreviaciones para clases de caracteres

Para encontrar cualquier cifra decimal, estas dos expresiones son equivalentes:

```
/[0-9]/
/\d/
```

Otras dos abreviaciones son:

- `\w` encuentra cualquier dígito, letra, o guión bajo `_`.
- `\s` encuentra cualquier carácter espacio-en-blanco (character whitespace), como son un espacio, un tabulado y un retorno de carro. Todas las abreviaciones precedentes, también tienen una forma negada. Para ello, se pone la misma letra en mayúsculas:

```
/\D/ # busca cualquier carácter que no sea un número
/\W/ # busca cualquier carácter que no sea una letra o guión bajo
/\S/ # busca un carácter que no sea un espacio en blanco.
```

Tabla resumen

expresión	significado
`.`	cualquier caracter
`[]`	especificación por rango. P.ej: `[a-z]` , una letra de la a, a la z
`\w`	letra o número; es lo mismo que `[0-9A-Za-z]`
`\W`	cualquier carácter que no sea letra o número

`\s`	carácter de espacio; es lo mismo que `[ \t\n\r\f]`
`\S`	cualquier carácter que no sea de espacio
`\d`	número; lo mismo que `[0-9]`
`\D`	cualquier carácter que no sea un número
`\b`	retroceso (0x08), si está dentro de un rango
`\b`	límite de palabra, si NO está dentro de un rango
`\B`	no límite de palabra
`*`	cero o más repeticiones de lo que le precede
`+`	una o más repeticiones de lo que le precede
`$`	fin de la línea
`{m,n}`	como menos m, y como mucho n repeticiones de lo que le precede
`?`	al menos una repetición de lo que le precede; lo mismo que `{0,1}`
`()`	agrupar expresiones
||	operador lógico O, busca lo de antes o lo después

Si no se entiende alguna de las expresiones anteriores, lo que hay que hacer es probar. Por ejemplo, veamos el caso de `|`. Supongamos que buscamos la palabra `gato` o la palabra `perro`:

```
/gato|perro/
```

El `|` es un "O lógico": se busca la palabra de la izquierda o la palabra de la derecha.

Una búsqueda con éxito

Cualquier búsqueda sucede con éxito o fracasa. Empecemos con el caso más simple: el fallo. Cuando intentas encontrar un string mediante un patrón, y el string no se encuentra, el resultado siempre es `nil` (nil = nada).

```
/a/.match("b") # nil
```

Sin embargo, si la búsqueda tiene éxito se devuelve un objeto `MatchData`. Este objeto tiene un valor 'true' desde el punto de vista booleano, y además almacena la información de lo encontrado: donde empieza (en qué carácter del string), qué porción del string ocupa,…Para poder usar esta información, hace falta almacenarla primero. Veamos un ejemplo donde buscamos un número de teléfono dentro de un string:

```
string = "Mi número de teléfono es (123) 555-1234."
num_expr = /\(\d{3}\)\s+(\d{3})-(\d{4})/ # expresión regular
m = num_expr.match(string)               # almacenamos búsqueda
unless m
  puts "No hubo concordancias."
  exit
end
print "El string de la búsqueda es: "
puts m.string    # string donde se efectúa la búsqueda
print "La parte del string que concuerda con la búsqueda es: "
puts m[0]        # parte del string que concuerda con nuestra búsqueda
puts "Las tres capturas:"
3.times do |index|
  # m.captures[index] - subcadenas encontradas (subcaden = () en la expresión)
```

```
  puts "Captura ##{index + 1}: #{m.captures[index]}"
end
puts "Otra forma para poner la primera captura: "
print "Captura #1: "
puts m[1] # cada número corresponde a una captura
```

la salida es:

```
El string de la búsqueda es: Mi número de teléfono es (123) 555-1234.
La parte del string que concuerda con la búsqueda es: (123) 555-1234
Las tres capturas:
Captura #1: 123
Captura #2: 555
Captura #3: 1234
Otra forma de poner la primera captura
Captura #1: 123
```

Para analizar la expresión regular, hay que prestar atención a cómo están agrupadas las búsquedas entre paréntesis:

```
num_expr = /\(((\d{3})\)\s+(\d{3})-(\d{4})/
```

- `\(((\d{3})\)` busca un grupo de tres números `(\d{3})`, entre dos paréntesis `\(…\)`
- `\s+` espacio en blanco una o varias veces
- `(\d{3})` tres números
- `-` signo menos
- `(\d{4})` cuatro números

Más malabares con strings

Hay muchos métodos en la clase `String` (no hay que memorizarlos todos; para algo está la documentación) como:

- `reverse`, que invierte los caracteres de un string
- `length`, que nos dice el número de caracteres de un string, incluyendo los espacios en blanco.
- `upcase`, que pone todos los caracteres en mayúsculas
- `downcase`, que pone todos los caracteres en minúsculas
- `swapcase`, pone los caracteres mayúsculas en minúsculas y los minúsculas en mayúsculas
- `capitalize`, pone el primer caracter del string en mayúsculas, y los demás en minúsculas
- `slice`, da una parte de un string

Los métodos `upcase`, `downcase`, `swapcase` y `capitalize` tienen su correspondiente método bang, que modifican el string (`upcase!`, `downcase!`, `swapcase!`, y `captalize!`). Si no necesitas el string original, es bueno usarlo, por que ahorrarás memoria; sobretodo si el string es largo.

Cada vez que se se asigna a una variable un string, se crea un nuevo objeto String. ¿Cómo es administrada la memoria en los strings? ¿Hay una porción separada para ellos? La clase String tiene más de 75 métodos. Leyendo la Guía de Uso de Ruby (Ruby User's Guide), dice "no tenemos en cuenta la memoria ocupada por un string. Prescindimos de cualquier administración de memoria". Para saber todos los métodos que tiene un String:

- String.methods, da una lista de todo los métodos que tiene la clase String.
- String.methods.sort (sort=ordenar), da una lista ordenada alfabéticamente de todos los métodos.
- String.instance_methods.sort, da una lista ordenada de todo los métodos de instancia (se explicará más adelante) que tiene un String.
- String.instance_methods(false).sort, muestra una lista ordenada de todos los métodos que pertanezcan exclusivamente a los Strings, pero no a las clases de las que desciende.

Comparando dos cadenas

Los strings tienen distintos métodos para comparar su igualdad. El más común de ellos es `==`. Otro método es `String.eql?`, que devuelve el mismo resultado que `==`. Y por último está String.equal?, que comprueba si dos strings son el mismo objeto. Veamos el siguiente ejemplo:

```
def compara_strings(s1, s2, s3)

#comprobamos si el contenido es igual
   if s1 == s2
     puts 'Ambos strings tienen el mismo contenido'
   else
     puts 'Ambos strings NO tienen el mismo conenido'
   end

   if s1.eql?(s2)
     puts 'Ambos strings tienen el mismo contenido'
   else
     puts 'Ambos strings NO tienen el mismo conenido'
   end

=begin
  ahora comprobamos si ambos objetos son iguales:
  dos objetos diferentes
  pueden tener el mismo contenido
=end

   if s1.equal?(s2)
     puts 'Ambos strings son el mismo objeto'
```

```
    else
      puts 'Ambos strings NO son el mismo objeto'
    end

    if s1.equal?(s3)
      puts 'Ambos strings son el mismo objeto'
    else
      puts 'Ambos strings NO son el mismo objeto'
    end

end

 string1 = 'Jonathan'
 string2 = 'Jonathan'
 string3 = string1

compara_strings(string1,string2,string3)
```

Ejercicio

Dado un string, invertirlo palabra por palabra (en vez de letra por letra).

Solución

Se puede usar String.split que nos da un array formado por las palabras del string. La clase Array tiene un método reverse; de tal forma que puedes revertir el array antes de juntarlo para hacer un nuevo string:

```
palabras = 'Tutorial de Ruby - fácil, sencillo y con fundamento'
puts palabras.split(" ").reverse.join(" ")
```

Condiciones

if,else

En Ruby, `nil` y `false` significan falso, todo lo demás (incluyendo `true`, `0`) significan verdadero. En Ruby, `nil` es un objeto: por tanto, tiene sus métodos, y lo que es más, puedes añadir los métodos que se quieran.

Veamos un ejemplo de `if,else`:

```
xyz = 5
if xyz > 4
  puts 'La variable xyz es mayor que 4'
  puts 'Puedo poner más instrucciones dentro del if'
  if xyz == 5
    puts 'Se puede anidar un bloque if,else,end dentro de otro'
  else
    puts "Parte del bloque anidado"
  end
else
  puts 'La variable xyz no es mayor que 4'
  puts 'También puedo poner múltiples sentencias'
end
```

elsif

`else` se ejecutaba si la condición en `if` no se cumplía. Para poder tomar más decisiones, en función del valor de la variable, se usa `elsif`:

```
#usando if,else anidados

puts 'Hola, cuál es tu nombre?'
STDOUT.flush
nombre = gets.chomp
puts 'Hola, ' + nombre + '.'

if nombre == 'Mojamuto'
  puts 'Pedazo de nombre!!!'
else
  if name == 'Enjuto'
    puts '...este nombre no es moco de pavo...'
  end
end
```

```
#usando elsif

puts 'Hola, cuál es tu nombre?'
STDOUT.flush
nombre = gets.chomp
puts 'Hola, ' + nombre + '.'

if nombre == 'Mojamuto'
  puts 'Pedazo de nombre!!!'
elsif nombre == 'Enjuto'
  puts '...este nombre no es moco de pavo...'
end
```

```
#otra modificación, usando el || ("o" lógico)
```

```
puts 'Hola, cuál es tu nombre?'
STDOUT.flush
nombre = gets.chomp
puts 'Hola, ' + nombre + '.'

if nombre = 'Mojamuto' || nombre = 'Enjuto'
  puts 'Pedazo de nombre!!!'
end
```

Además de la igualdad, existen otros operadores condicionales:

operador	significado
==	igual
!=	distinto
>=	mayor o igual que
<=	menor o igual que
>	mayor que
<	menor que

case

Esta instrucción es muy parecida al `if:` se crean una serie de condiciones, y se ejecuta la primera condición que se cumpla. Por ejemplo:

```
xyz = 10

if xyz % 2 == 0
  puts 'La variable xyz es par'
else
  puts 'La variable xyz es impar'
end
```

es equivalente a:

```
xyz = 10

par = case
  when xyz % 2 == 0 then true
  when xyz % 2 != 0 then false
end

puts par
unless
```

Ruby tiene una forma contraria al `if:` la instrucción unless. Y digo contraria, por que en `if` se ejecutaba el bloque `do … end` si se cumplía la condición; con `unless` se ejecutará el bloque mientras NO se cumpla.

```
nombre = 'Pepe'
unless nombre == 'Enjuto'
  puts 'Ese nombre no tiene arte ninguno'
end

=begin
  Si el nombre no es Enjuto,
```

```
    siempre se ejecutará el bloque.
=end
```

Ejercicios

Escribe un método que pregunte por un año, y sea capaz de:

1. Decir si es, o no es bisiesto.
2. Poner la cantidad de minutos que tiene el año.

Bucles

while

Se pueden hacer bucles (bucle = algo que se repite) con la instrucción `while` :

```
# Loops
var = 0
while var < 10
  puts var
  var += 1
end
```

times

Este es un ejemplo de cómo en Ruby es todo un objeto, inclusive los números. El método times necesita dos cosas:

1. un número entero, del cuál es el método
2. un bloque

Lo que hace times es iterar el bloque ese "número" de veces.

```
10.times do |num|
  puts num
end
```

Fijarse, que la variable num va de 0 a 9; por lo tanto, el bloque se itera 10 veces.

Números aleatorios

Ruby tiene con un generador de números aleatorios: el método `rand`. Usando `rand` se obtiene un número aleatorio x, tal que 0 <= x < 1. Si se le da un parámetro, por ejemplo `rand(5)`, entonces se obtiene un número entero entre 0 y 4 (ambos incluidos).

```
def carta_aleatoria
  palos = %w{ corazones treboles picas diamantes}
  numero = %w{ 1 2 3 4 5 6 7 8 9 10 J Q K }

  #Quiero una carta aleatoria que tiene:
  #  -un palo aleatorio
  #  -un número aleatorio

  #palo aleatorio
  num = palos.length
  palo_aleat = rand(num)

  #numero aleatorio
  num_aleat = rand(numero.length)

  puts numero[num_aleat] + ' de ' + palos[palo_aleat]
end

#una carta aleatoria
carta_aleatoria

#10 cartas aleatorias
10.times do
  carta_aleatoria
end

#NOTA: la variable del bucle,
#como no se usa en el bloque
#no se define.
```

Clases y Objetos

Desde hace tiempo el estilo de programación funcional (que se usa por ejemplo en el lenguaje C) es usado para programar. En este tipo de programación, hay que centrarse en los pasos para realizar la tarea, y nos olvidamos de como se manejan los datos. Sin embargo, en la programación orientada a objetos, los objetos son los agentes, el universo de tu programa: se presta atención a la estructura de los datos. Cuando se diseña una clase, se piensa en los objetos que serán creados por esa clase: en las cosas que podrá hacer ese objeto, y las características que lo definen.

Un objeto es un contenedor de datos, que a su vez controla el acceso a dichos datos. Asociados a los objetos está una serie de variables que lo definen: sus atributos. Y también un conjunto de funciones que crean un interfaz para interactuar con el objeto: son los métodos. Un objeto es una combinación de estado y de métodos que cambian ese estado.

Una clase es usada para construir un objeto. Una clase es como un molde para objetos. Y un objeto, una instancia de la clase. Por ejemplo, se puede usar la clase `Button` para hacer docenas de botones, cada botón con su propio color, tamaño, forma,…

```
# Nuestra primera clase

# define la clase Perro
class Perro

  # método inicializar clase
  def initialize(raza, nombre)
    # atributos
    @raza = raza
    @nombre = nombre
  end

  # método ladrar
  def ladrar
    puts 'Guau! Guau!'
  end

  # método saludar
  def saludar
    puts "Soy un perro de la raza #{@raza} y mi nombre es #{@nombre}"
  end
end

# para hacer nuevos objetos,
# se usa el método new
d = Perro.new('Labrador', 'Benzy')
puts d.methods.sort
puts "La id del ojbeto es #{d.object_id}."

if d.respond_to?("correr")
  d.correr
else
  puts "Lo siento, el objeto no entiende el mensaje 'correr'"
end

d.ladrar
d.saludar

# con esta variable, apuntamos al mismo objeto
d1 = d
d1.saludar

d = nil
d1.saludar
```

El método `new` se usa para crear un nuevo objeto de la clase `Perro`. Los objetos son creados en el momento y el espacio de memoria donde se guardan, se asigna a una variable, en este caso la variable `d`, que se conoce como variable de referencia.

Un método recién creado no es un espacio en blanco: un objeto recién creado, puede responder un montón de mensajes. Para ver la lista de esos mensajes o métodos de forma ordenada (.sort):

```
puts d.methods.sort
```

El resultado es una lista de todos los mensajes o métodos que el objeto recién creado puede responder. De todos esos métodos, los `object_id` y `respond_to?` son importantes.

object_id, respond_to?

Cada objeto en Ruby tiene un único número asociado con él. Se puede ver dicho número mediante el método `object_id`. En nuestro ejemplo:

```
puts "El número que identifica al objeto denotado por la variable d es #{d.object_id}."
```

Se puede conocer de antemano, si un objeto será capaz de responder a un mensaje; o dicho de otra forma, si un objeto posee cierto método. Para ello se usa `respond_to?`. En nuestro ejemplo:

```
if d.respond_to?("correr")
  d.correr
else
  puts "Lo siento, el objeto no entiende el mensaje 'correr'"
end
```

class, instance_of?

Puedes saber a qué clase pertenece un objeto mediante el método `class`. En nuestro ejemplo si ponemos:

```
d = Perro.new('Alsatian', 'Lassie')
puts d.class.to_s # obtenemos Perro
```

`instance_of?` nos devuelve true si un objeto es instancia de una clase determinada. Por ejemplo:

```
num = 10
puts (num.instance_of? Fixnum) # true
```

La clase Class

Las clases en Ruby son instancias de la clase `Class`. Cuando se define una nueva clase (p.e. `class NombreClase … end`), se crea un objeto de la clase `Class` y es asignado a una constante (en este caso NombreClase). Cuando se usa `NombreClase.new` para construir un nuevo objeto, se usa el método de la clase `Class` para crear nuevas instancias; y después se usa el método inicializador de la propia clase NombreClase: la construcción y la inicialización de un objeto son cosas distintas, y pueden modificarse.

Constructores literales

Significa que se puede usar una notación especial, en vez de usar `new` para crear un nuevo objeto de esa clase. Las clases que un constructor literal puede crear, están en la siguiente tabla: cada vez que usas uno de estos constructores, creas un nuevo objeto.

Clase	Constructor Literal	Ejemplo
String	`'` ó `"`	"nuevo string" o 'nuevo string'
Símbolo	`:`	:símbolo o :"símbolo con espacios"
Array	`[ ]`	[1, 2, 3, 4, 5]
Hash	`{ }`	{"Nueva Yor" => "NY", "Oregon" => "OR"}
Rango	`..` ó `…`	0…10 ó 0..9
Expresiones regulares	`//`	/([a-z]+)/

Por esto, y aunque a veces no lo parezca, siempre estamos creando objetos. En Ruby, todo es un objeto.

Reciclado de basura

La instrucción:

```
d = nil
```

hace que d apunte a nil, lo que significa que no se refiere a nada. Si después de eso, añado:

`Ruby d1 = nil` Ruby

entonces el objeto `Perro` dejará de estar apuntado por las variables y será objetivo del reciclado de basura. El reciclado de basura de Ruby es del tipo marcar-y-borrar (mark-and-sweep):

- en la fase "mark" el programa que recicla la memoria, verifica si el objeto está en uso. Si un objeto es apuntado por una variable, podría ser usado, y por tanto ese objeto se marca para ser conservado.
- si no hay variable que apunte al objeto, entonces el objeto no es marcado. Y en la fase "sweep" se borran los objetos en desuso para liberar memoria de modo que pueda volver a ser utilizada por el intérprete de ruby.

Ruby usa un mark-and-sweep conservador: no hay garantía de que un objeto sea eliminado por el colector, antes de que termine el programa. Si almacenas algo en un array, y se mantiene el array, todo dentro del array es marcado. Si almacenas algo en una constante o variable global, entonces se marca para siempre.

Métodos de clase

La idea de los métodos de clase es mandar el mensaje a la clase, en vez de una de sus instancias. Los métodos de clase se usan porque algunas operaciones que pertenecen a una clase, no pueden ser realizadas por sus instancias. `new` es un buen ejemplo. La tarea de crear nuevos objetos sólo la puede hacer la clase; los objetos no pueden crearse a sí mismos.

Accesor (métodos de acceso)

Los accesores permiten el acceso a los atributos del objeto.

```ruby
# SIN accesores

class Cancion
  def initialize(titulo, artista)
    @titulo = titulo
    @artista = artista
  end
  def titulo
    @titulo
  end
  def artista
    @artista
  end
end

cancion = Cancion.new("Brazil", "Ivete Sangalo")
puts cancion.titulo
puts cancion.artista

# CON accesores

class Cancion
  def initialize(titulo, artista)
    @titulo = titulo
    @artista = artista
  end

  # accesor de lectura
  attr_reader :titulo, :artista

  # accesor de escritura
  # attr_writer :titulo

  # accesor de escritura y lectura
  # attr_accessor :titulo
end

cancion = Cancion.new("Brazil", "Ivete Sangalo")
puts cancion.titulo
puts cancion.artista
```

Ficheros: lectura/escritura

Veamos como se puede leer/escribir un fichero con un ejemplo:

```
# 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 retorno de carro
  f2.puts "Por que la vida \n puede ser maravillosa"
end
```

El método `File.open` puede abrir el fichero de diferentes formas:

- `'r'` sólo-lectura, comienza al principio del fichero.
- `'r+'` lectura/escritura, comienza al principio del fichero.
- `'w'` sólo-escritura, crea un nuevo fichero o elimina el contenido del fichero, para empezar de cero.

Hay que consultar la documentación para ver una lista completa de los posibles modos. `File.open` abre un nuevo fichero si no hay un bloque; pero si usa un bloque, entonces el fichero será el argumento del bloque, y se cerrará automáticamente cuando termine el bloque. Si no hay bloque, el fichero no se cierra de forma automática, y siempre, siempre hay que cerrar un fichero: en el caso de un fichero abierto para escritura, si no se cierra, podrían perderse datos.

El método `readline` de `File` copia el fichero línea por línea dentro de un array. Ambos métodos, open y readlines pertenecen a la clase `IO`, de la cual desciende la clase `File`; y por tanto, son heredados.

Manejando directorios

El módulo Find hace una búsqueda descendente en los directorios de un conjunto de rutas/directorios. Si el argumento es un directorio, entonces el resultado será el directorio, y todos los ficheros y subdirectorios que le pertenecen.

```
require 'find'

# muestra la ruta ./
# que es el directorio de Ruby
Find.find('./') do |f|
    type = case
    # si la ruta es un fichero -> F
        when File.file?(f) then "F"
        # si la ruta es un directorio -> D
        when File.directory?(f) then "D"
        # si no sabemos lo que es -> ?
        else "?"
    end
  # formatea el resultado
  puts "#{type}: #{f}"
end
```

Acceso aleatorio

Por acceso aleatorio, se entiende, empezar a leer el fichero en cualquier parte. Supongamos 'aleatorio.rb':

aleatorio.rb

puts 'Surfing USA' Ahora queremos leer el fichero a partir de la palabra "USA":

```
f = File.new("aleatorio.rb")

f.seek(12, IO::SEEK_SET)
print f.readline
f.close
```

El método seek de la clase IO, busca una posición dada por el primer parámetro, de la forma indicada por el segundo parámetro. Las posibles formas son:

- `SEEK_CUR` - busca (seek) desde el primer parámetro, un número entero, hasta la posición actual.
- `SEEK_END` - busca desde el parámetro dado, hasta el final del fichero.
- `SEEK_SET` - busca la posición absoluta dada por el parámetro :: es el operador de alcance

¿Permite Ruby la serialización de objetos?

Java tiene la habilidad de serializar objetos: te permite almacenarlos en un fichero y luego reconstruirlos cuando es necesario. Ruby llama a este tipo de serialización marshalling.

Salvar un objeto y algunos o todos de sus componentes, se hace mediante el método `Marshall.dum`. Después, se puede recuperar el objeto usando `Marshall.load`. Se profundizará más adelante en el tema.

Usando librerías

Cuando escribes tus primeros programas, se tiende a guardar todo el código en un mismo fichero. Pero a medida que creces como programador, tus programas también lo hacen, y en algún momento te darás cuenta que tener un único fichero contiendo todo el código, es poco práctico. Es mucho más fácil dividir el código en grupos y colocar cada grupo en un fichero: una librería (en español sería biblioteca de programas o módulos, no de libros) es un fichero que contiene métodos y clases para su uso a posteriori. Para poder usar estas librerías, necesitas de los métodos `require` y `load`.

require

El método require lee una única vez el fichero especificado:

hola.rb

```
puts "Hola a todos!"
```

Al usar varias veces require:

```
require 'hola' # Hola a todos!
require 'hola' #
require 'hola' #
```

Si volvemos a usarlo, omite su lectura: todas las clases y métodos del fichero, ya están almacenados en memoria. Por tanto nos ahorramos el tiempo de otra lectura. Hay que fijarse en un detalle a la hora de usar `require`:

```
require 'hola'      # bien
```

`require` da por hecho que la extensión del fichero es .rb, buscando el fichero 'hola.rb'. Aunque no fuese .rb, Ruby seguiría buscando entre las demás extensiones hasta encontrar un fichero del nombre 'hola'.

load

El método `load` lee el fichero indicado tantas veces como aparezca la instrucción. ¿Difícil? No tanto, veamos un ejemplo:

```
load 'hola.rb' # Hola a todos!
load 'hola.rb' # Hola a todos!
load 'hola.rb' # Hola a todos!
```

Cada vez que se lee, todas los métodos del fichero se vuelven a leer, y están disponibles para su uso. Cuando se usa load es porque se espera que algo haya cambiado en el fichero.

RAILS: en Rails se suele usar `load`. Por ejemplo: mientras se desarrolla, lo que significa estar probando la aplicación y haciendo cambios al mismo tiempo, los cambios se actualizan al releer el fichero. Usar `require` no tendría el mismo efecto, puesto que el fichero se ha leído una vez.

Otro ejemplo

moto.rb

```
class Moto

  def initialize(marca, color)
    # Atributos (variables del objeto)
    @marca = marca
    @color = color
  end

  def arrancar
    if (@estado_motor)
      puts 'Motor encendido'
    else
      @estado_motor = true
      puts 'Arrancando el motor'
    end
  end

end
```

Si ahora queremos usar la clase Moto, en otro fichero:

```
require 'moto.rb'
m = Moto.new('Yamaha', 'rojo')
m.arrancar
```

Herencia de clases

La herencia de clases es una relacción entre dos clases. La ventaja de la herencia es que las clases que en una jerarquía están en un nivel inferior, heredan las características de las clases de niveles superiores; y además, pueden añadir sus propias características.

Por ejemplo: todos los gatos son mamíferos. Si todos los mamíferos respiran, la clase gato por descender de la clase mamífero hereda esta característica: los gatos respiran.

Esto puede programarse así:

```
class Mamifero
  def respirar
    puts 'inspirar, espirar'
  end
end

# el símbolo < indica que
# Gato es una subclase de Mamifero

class Gato < Mamifero
  def maullar
    puts 'Miaaaaaaaaaaau'
  end
end

cribas = Gato.new
cribas.respirar
cribas.maullar
```

Aunque no se especificó que los gatos puedan respirar, todos los gatos heredarán esa característica de la clase `Mamifero`, ya que el gato es una subclase de los mamíferos. En el argot, `Mamifero` es la super-clase o clase padre, y `Gato` es la subclase, o clase hija. Esto es una ventaja para el programador: los gatos tienen la capacidad de respirar, sin haberlo implementado.

En Ruby, como se mostró en este esquema, la clase `Object` es la madre de todas las clases en Ruby; por lo tanto, sus métodos están disponibles en todos los objetos, excepto aquellos que se han sobrescrito.

Sobrescritura de métodos (method overriding)

Habrá situaciones donde las propiedades de una super-clase no deberían ser heredadas por una subclase en particular. Por ejemplo, las aves generalmente saben volar, pero los pingüinos son una subclase de `Ave`, y no vuelan:

```
class Ave
  def asear
    puts 'Me estoy limpiando mis plumas.'
  end

  def volar
    puts 'Estoy volando.'
  end
end

class Pinguino < Ave
  def volar
    puts 'Lo siento, no soy capaz de volar.'
  end
end
```

```
p = Pinguino.new
p.asear
p.volar
```

Se ha sobrescrito el método volar. La gran ventaja que aporta el uso de la herencia de clases, se llama programación diferencial: vamos de lo más general a lo más particular, añadiendo y modificando donde sea necesario.

Los dos ejemplos anteriores son traducciones de la guía online "Ruby User's Guide".

Super

```
class Bicicleta
  attr_reader :marchas, :ruedas, :asientos # se hablará de attr_reader
  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
```

Cuando uno usa super dentro de un método, Ruby manda un mensaje a la clase madre del objeto al que pertence el método, buscando un método con el mismo nombre. Si:

- se invoca con una lista vacía de argumentos (como este caso), super ó `super()`, no se pasan argumentos al método de la clase madre.
- se invoca con argumentos, `super(a, b, c)`, se mandand los argumentos a, b, c.

En este caso, se usa super en el método `initialize` de Tandem, lo que provoca el uso del initialize de Bicicleta para crear instancias de Tandem. La salida es:

```
2
2
2
1
2
1
```

RAILS: la herencia de clases es una de las claves en el desarrollo de RAILS

Modificando clases

En Ruby, las clases nunca están cerradas: siempre se pueden añadir métodos a una clase. Esto es válido tanto para las clases que escribas, como para las que ya están incluidas con el intérprete. Todo lo que hay que hacer, es continuar con la definición de la clase:

```
require 'moto'
m = moto.new('Yamaha', 'rojo')
m.arrancar

class Moto
  def informe_moto
    puts 'El color de la moto es ' + @color
    puts 'La marca de la moto es ' + @marca
  end
end

m.informe_moto
```

Ahora añadamos un método a la clase `String`:

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

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

Si se escribe un nuevo método que conceptualmente pertenece a la clase original, se puede reabrir el fichero de la clase, y añadir el método a la definición de la clase. Esto hay que hacerlo cuando el método es de uso frecuente, y se está seguro que no entrará en conflicto con otros métodos definidos en otras librerías que se usen más adelante.

Si el método no será usado frecuentemente, o no se quiere tomar el riesgo de modificar la clase después de su creación, crear una subclase (ver Herencia) es la mejor opción. Una clase puede sobrescribir los métodos de la clase de la que desciende. Y es más seguro, por que la clase original permanece intacta.

Sobrecarga de métodos (methods overloading)

Las clases en Ruby sólo pueden tener un método con un nombre dado. Para tener métodos "distintos" con el mismo nombre, se puede jugar con el número de argumentos:

```
# Un cuadrado se puede definir de dos formas:
#   Cuadrado.new([x_sup, y_izq], ancho, alto)
#   Cuadrado.new([x_sup, y_izq], [x_inf, y_der])

class Cuadrado
  def initialize(*args) # * implica número variable de argumentos
    if args.size < 2 || args.size > 3
      puts 'ERROR: Este método toma dos o tres argumentos'
    else
      if args.size == 2
        puts 'Dos argumentos'
      else
        puts 'Tres argumentos'
      end
    end
```

```
  end
end

Cuadrado.new([10,23], 4, 10)            # Tres argumentos
Cuadrado.new([10,23], [14,13])            # Dos argumentos
Cuadrado.new([10,23], [14,13],4,10)            # ERROR: Este método toma dos o tres argumentos
```

El programa está incompleto, pero es suficiente para ver cómo se puede conseguir la sobrecarga de métodos.

Congelando objetos

Los objetos inmutables son aquellos que no pueden cambiar de estado después de ser creados. Las propiedades por las que destacan son:

- ser thread-safe. Los threads no pueden corromper lo que no pueden cambiar.
- facilitar el implementar la encapsulación: si parte del estado de un objeto es almacenado dentro un objeto inmutable, entonces los métodos modificadores pueden leer el estado de dicho objeto, sin miedo a que modifiquen su estado.
- ser índices en los hashes inmutables, ya que no pueden cambiar.

En Ruby, la mutabilidad es una propiedad de los objetos, no de una clase entera. Cualquier objeto (o instancia) se puede volver inmutable usando `freeze`.

freeze

El método `freeze` (congelar) evita que un objeto pueda modificarse, convirtiéndolo en una constante. Después de "congelar" el objeto, cualquier intento de modificarlo da como resultado un `TypeError`.

```
str = 'Un simple string'
str.freeze # congelamos el string

# se intenta modificar (begin)
# y en caso de error (rescue)
# se lanza un mensaje. Ver Excepciones.

begin
  str << 'Intento de modificarlo'
rescue => err
  puts "#{err.class} #{err}"
end
```

La salida es - TypeError can't modify frozen string.

Sin embargo, freeze funciona con las referencias, no con las variables: esto significa que si creamos un objeto nuevo, y sobreescribimos la variable, este se podrá modificar:

```
str = 'string original - '
str.freeze
str += 'añadido a posteriori'
puts str
```

La salida es - 'Original string - añadido a posteriori'

El objeto original no cambió. Sin embargo, la variable `str` se refiere a un nuevo objeto. El método `frozen?` nos dice si un objeto está congelado o no.

Serializando Objetos

Java es capaz de serializar objetos: puede almacenarlos, para luego reutilizarlos cuando sea necesario. Ruby tiene también esta capacidad, pero la llama **marshaling**. Veamos un ejemplo en el que a partir de una clase, creamos una serie de objetos que almacenamos y luego recuperamos:

- La clase "personaje.rb":

```
class Personaje
  def initialize(vida, tipo, armas)
    @vida = vida
    @tipo = tipo
    @armas = armas
  end
  attr_reader :vida, :tipo, :armas
end
```

- Creamos los objetos y los guardamos en un fichero usando `Marshal.dump`:

```
require 'personaje'
p1 = Personaje.new(120, 'Mago', ['hechizos', 'invisibilidad'])
puts p1.vida.to_s+' '+ p1.tipo+' '
p1.armas.each do |w|
  puts w + ' '
end

File.open('juego', 'w+') do |f|
  Marshal.dump(p1, f)
end
```

- Usamos `Marshal.load` para recuperarlos:

```
require 'personaje'
File.open('juego') do |f|
  @p1 = Marshal.load(f)
end

puts @p1.vida.to_s + ' ' + @p1.tipo + ' '
@p1.armas.each do |w|
  puts w + ' '
end
```

`Marshal` únicamente serializa estructuras de datos; no puede serializar código (como hacen los objetos `Proc`), o recursos utilizados por otros procesos (como conexiones a bases de datos). `Marshal` da un error cuando se intenta serializar un fichero.

Control de Acceso y Accesores

En Ruby, la única forma de cambiar el estado de un objeto, es invocando uno de sus métodos: si controlas el acceso a laso métodos, controlarás el acceso a los objetos. Una buena regla, es cerrar el acceso a los métodos que puedan dejar al objeto en un estado no válido.

Los tres niveles de acceso

- **public** - los métodos públicos (public) pueden ser usados por cualquiera; no hay un control de acceso.
- **protected** - los métodos protegidos (protected) pueden ser usados únicamente por objetos de la misma clase y subclases, a las que pertenece el método; pero nunca por el propio objeto. Por así decirlo, el método sólo lo pueden usar los otros miembro de la familia.
- **private** - los métodos privados (private) sólo pueden ser usado por el propio objeto. Técnicamente, se dice que el receptor del método siempre es el mismo: self. El control de acceso se determina dinámicamente, a medida que el programa transcurre. Se obtiene una violación de acceso siempre que se intenta ejecutar un método no público.

```
class ControlAcceso
  def m1    # este método es público
  end
  protected
    def m2  # este método es protegido
    end
  private
    def m3  # este método es privado
    end
    def m4
    end
end

ca = ControlAcceso.new
ca.m1
ca.m2
ca.m3
```

La privacidad de los métodos, también se pueden especificar de esta forma:

```
class ControlAcceso
  def m1    # este método es público
  end
  def m2  # este método es protegido
  end
  def m3  # este método es privado
  end
  def m4  # este método es privado
  public :m1
  protected :m2
  private :m3, :m4
end

ca = ControlAcceso.new
ca.m1
ca.m2
ca.m3
```

protected

Tal vez el nivel de acceso protegido (protected) sea un poco lioso de entender. Es mejor verlo con un ejemplo:

```
class Persona
  def initialize(edad)
    @edad = edad
  end
  def edad
    @edad
  end
  def comparar_edad(op)  # op = otra persona
    if op.edad > edad
      'La edad de la otra persona es mayor.'
    else
      'La edad de la otra persona es la misma o menor.'
    end
  end
  protected :edad
end

pedro = Persona.new(15)
almudena = Persona.new(17)
puts Pedro.comparar_edad(almudena) # La edad ... es mayor
```

El objeto que hace la comparación (`pedro`) necesita preguntar al otro objeto (`almudena`) su edad, lo que significa que ejecute su método edad. Por eso el nivel de acceso es protegido y no privado: al estar protegido `pedro` puede usar el método de `almudena` .

La excepción viene cuando `pedro` se pregunta a sí mismo la edad, por ser un método protegido, esto no será posible. self no puede ser el receptor de un método protegido.

Por ejemplo:

```
puts Pedro.edad #da error
```

Excepciones

Una excepción es un tipo especial de objeto de la clase Exception. Lanzar una excepción significa parar la ejecución de un programa para salir del mismo o para tratar con el problema que se ha producido. Para tratar el problema hace falta `raise`; de no ser así, el programa termina y avisa del error. Lo que hará `raise` (lanzar), será lanzar una "excepción" para manejar el error. Ruby tiene una serie de clases, `Exception` y sus hijas, que ayudan a manejar los errores que pueden ocurrir.

```
def lanzar_excepcion
  puts 'Estoy antes del raise'
  raise 'Se ha producido un error' # lanza una excepción con el mensaje entre ''
  puts 'Estoy despues del raise'
end

lanzar_excepcion
```

El método `raise` procede del módulo `Kernel`. Por defecto, `raise` crea una excepción de la clase `RuntimeError`. Para lanzar una excepción de una clase específica, se puede poner el nombre de la clase como argumento de `raise`.

```
def inverse(x)
  raise ArgumentError, 'El argumento no es numerico' unless x.is_a? Numeric
  1.0 / x
end
puts inverse(2)
puts inverse('patata') # da un error que es manejado por raise
```

Hay que recordar que los métodos que actúan como preguntas, se les pone un `?` al final: `is_a?` pregunta al objeto cuál es su tipo. Y unless cuando se pone al final de una instrucción, significa que NO se ejecuta cuando la expresión a continuación es verdadera.

Manejando una excepción

Para tratar una excepción, se pone el método que puede causar el error dentro de un bloque `begin…end`. Dentro de este bloque, se pueden poner varios `rescue` para cada tipo de error que pueda surgir:

```
def raise_and_rescue
  begin
    puts 'Estoy antes del raise'
    raise 'Un error ha ocurrido' # simulamos un error
    puts 'Estoy después del raise'
  rescue
    puts 'Estoy rescatado del error.'
  end
  puts 'Estoy después del bloque'
end

raise_and_rescue
```

La salida es:

```
Estoy antes del raise
Estoy rescatado del error.
Estoy después del bloque
```

Observar que el código interrumpido por la excepción, nunca se ejecuta. Una vez que la excepción es manejada (por el `rescue`), la ejecución continúa inmediatamente después del bloque `begin` fuente del error.

Al escribir `rescue` sin parámetros, el parámetro `StandardError` se toma por defecto. En cada rescue se pueden poner varias excepciones a tratar. En el caso de poner múltiples rescues:

```
begin
  #
rescue UnTipoDeExcepcion
  #
rescue OtroTipoDeExcepcion
  #
else
  # Otras excepciones
end
```

Ruby compara la excepción que produce el error, con cada rescue hasta que sea del mismo tipo; o sea una superclase de la excepción. Si la excepción no concuerda con ningún rescue, usar else se encarga de manejarla.

Para saber acerca del tipo de excepción, hay que mapear el objeto Exception a una variable usando rescue:

```
begin
  raise 'Test de excepcion'
rescue Exception => e
  puts e.message            # Test de excepción
  puts e.backtrace.inspect   # ["nombre de fichero:linea de la excepción"]
end
```

Si además de manejar la excepción, se necesita que se ejecute un código, se usará la instrucción ensure: lo que haya en ese bloque, siempre se ejecutará cuando el bloque `begin…end` termine.

Excepciones más comunes

He aquí algunas excepciones más comunes, con la causa que las origina y un ejemplo:

- `RuntimeError` - la excepción que se lanza por defecto. Ejemplo:

```
raise
```

- `NoMethodError` - el objeto no puede manejar el mensaje/método que se le envía. Ej:

```
string = 'patata'
string.multiplicarse
```

- `NameError` - el intérprete encuentra un identificador que no puede resolver ni como método, ni como variable. Ej:

```
a = variable_sin_definir
```

- `IOError` - lectura de un stream cerrado, escribir a un sistema de sólo lectura y operaciones parecidas. Ej:

```
STDIN.puts("No escribas a STDIN!")
```

```
Errno::error - errores relaccionado con el fichero IO. Ej:
File.open(-12)
```

- `TypeError` - un método recibe un argumento que no puede manejar. Ej:

```
a = 3 + "no puedo sumar un string a un número!"
```

- `ArgumentError` - causado por un número incorrecto de argumentos. Ej:

```
def m(x)
end
m(1,2,3,4,5)
```

Ejemplo

```
begin
  # Abre el fichero y lo lee
  File.open('origen.txt', 'r') do |f1|
    while line = f1.gets
      puts line
    end
  end

  # Crea un nuevo fichero y escribe en él
  File.open('destino.txt', 'w') do |f2|
    f2.puts "Creado por Satish"
  end
rescue Exception => msg
  # dispone el mensaje de error
  puts msg
end
```

Constantes Una constante es una referencia inmutable a un objeto; mientras que las variables sí se podían. Las constantes se crean cuando son asignadas por primera vez. En la actual versión de Ruby, reasignar una constante (intentar cambiar su valor) genera una advertencia, pero no un error.

Las constantes se ponen en mayúsculas:

```
CONST = 10
CONST = 20
```

Alcance de las constantes

- Las constantes definidas dentro de una clase o módulo pueden ser usadas en cualquier lugar dentro de la clase o módulo.
- Fuera de la clase o módulo, se pueden usar mediante el opoerador `::` precedido de una palabra que indique el módulo o clase apropiados.
- Las constantes definidas fuera de cualquier clase o módulo pueden ser usadas mediante el operador `::` pero sin palabra que lo preceda.
- Las constantes no pueden ser definidas dentro de un método.

```
CONST_EXTERNA = 99

class Const
    CONST = CONST_EXTERNA + 1
    def get_const
      CONST
    end
end

puts Const.new.get_const # 100
puts Const::CONST       # constante dentro de la clase Const
puts ::CONST_EXTERNA    # constante externa a toda clase
puts Const::NEW_CONST = 123
```

Repaso a los tipos de variables

```
# los nombres de las variables y métodos empiezan por minúsculas

$glob = 5             # las variables globales empiezan por $
class TestVar         # nombre de clase, empieza por mayúsculas
 @@cla = 6            # las variables de clase empiezan por @@
 CONST_VAL = 7        # constante: todo mayúsculas y/o _
 def initialize(x)    # constructor
  @inst = x           # variables de objeto empiezan por @
  @@cla += 1          # cada objeto comparte @@cla
 end
 def self.cla         # método de clase, lector de atributo
  @@cla
 end
 def self.cla=(y)     # método de clase, modificador de atributo"0@%0 @@cla = y
 end
 def inst             # método de objeto, lector
  @inst
 end
 def inst=(i)         # método de objeto, modificador
  @inst = i
 end
end

puts $glob
test = TestVar.new(3)
```

```
puts test.inspect     # da el ID del objeto y sus variables
TestVar.cla = 4
test.inst=8
puts test.inst
puts TestVar.cla
otro = TestVar.new(17)
# 'cla' se modifica cada vez
# que se crea un objeto
puts TestVar.cla
puts otro.inspect
```

Hashes y Símbolos

Un símbolo parece una variable, pero está precedido de dos puntos. Ejemplos:

```
:action
:line_tines
```

Los dos puntos se pueden interpretar como "la cosa llamada". Entonces `:id`, se interpreta como "la cosa llamada id". Los símbolos no contienen valores como las variables. Un símbolo es una etiqueta, un nombre, nada más.

Símbolos vs Strings

Un símbolo es el objeto más básico que puedes crear en Ruby: es un nombre y una ID interna. Los símbolos son útiles por que dado un símbolo, se refiere al mismo objeto en todo el programa. Por lo tanto, son más eficientes que los strings: dos strings con el mismo nombre, son dos objetos distintos. Esto implica un ahorra de tiempo y memoria.

```
puts "hola".object_id   # 21066960
puts "hola".object_id   # 21066730
puts :hola.object_id    # 132178
puts :hola.object_id    # 132178
```

Cada vez que se ha usado un string, se ha creado un objeto nuevo. Por tanto, ¿cuándo usar un string, y cuándo un símbolo?

- Si el contenido del objeto es lo importante, usa un string.
- Si la identidad del objeto es importante, usa un símbolo.

Ruby usa una tabla de símbolos interna con los nombres de las variables, objetos, métodos, clases… Por ejemplo, si hay un método con el nombre de `control_movie`, automáticamente se crea el símbolo `:control_movie`. Para ver la tabla de símbolos `Symbol.all_symbols`.

Como veremos a continuación, los símbolos son particularmente útiles para los hashes.

Hashes

Hashes, también conocidos como arrays asociativos, mapas o diccionarios, son parecidos a los arrays en que son una colección indexada de referencias a objetos. Sin embargo, mientras que en los arrays los índices son números, en los hashes se puede indexar con objetos de cualquier tipo: strings, expresiones regulares, etc.

Cuando se almacena un valor en un array, se dan dos objetos: el índice y el valor. A posteriori se puede obtener dicho valor, gracias al índice.

```
h = {'perro' => 'canino', 'gato' => 'felino', 'burro' => 'asno', 12 => 'docena'}
puts h.length   # 4
puts h['perro'] # 'canino'
puts h
puts h[12]
```

Comparados con los arrays, tenemos una ventaja significativa: se puede usar cualquier objeto como índice. Sin embargo,

sus elementos no están ordenados, y es fácil usar un hash como una pila o cola.

Los hashes tienen un valor por defecto. Este valor se devuelve cuando se usan índices que no existen: el valor que se devuelve por defecto es nil.

La clase Hash tiene muchos métodos que se pueden ver aquí.

Los símbolos como índices

Por las ventajas antes citadas, se usan los símbolos como índices:

```
persona = Hash.new
persona[:nombre] = 'Pedro'
persona[:apellido] = 'Picapiedra'

puts persona[:nombre]
```

que es equivalente a:

```
persona = {:nombre => 'Pedro', :apellido => 'Picapiedra'}

puts persona[:apellido]
```

La clase Time

La clase Time en Ruby tiene un extraordinario método para formatear su resultado, que es de gran utilidad a la hora de representar la hora de distintas formas. La clase `Time` de Ruby contiene un interface para manejar directamente las librerías escritas C sobre las horas.

El cero de los tiempos para Ruby, es el primer segundo GMT del 1 de Enero de 1970. Esto puede traer problemas a la hora de representar instantes anteriores a ese cero. La clase DateTime es superior a Time para aplicaciones astronómicas o históricas; sin embargo, para las aplicaciones normales, con usar Time es suficiente.

```
t = Time.now

puts t.strftime("%d/%m/%Y %H:%M:%S")
# strftime - formatear tiempo (stringfy time)
# %d - día (day)
# %m - mes (month)
# %Y - año (year)
# %H - hora en formato 24 horas (hour)
# %M - minuto
# %S - segundo (second)

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")
# %Z - zona horaria
```

self

En cada instante de la ejecución del programa, hay uno y sólo un self: el objeto que se está usando en ese instante.

Contexto del nivel superior

El contexto del nivel superior se produce si no se ha entrado en otro contexto, por ejemplo, la definición de una clase. Por la tanto, el término "nivel superior" se refiere al código escrito fuera de las clases o módulos. Si abres un fichero de texto y escribes:

```
x = 1
```

habrás creado una variable local en el nivel superior. Si escribes:

```
def m
end
```

habrás creado un método en el nivel superior: un método que no es definido como un método de una clase o módulo. Si nada más arrancar el intérprete, tecleas:

```
puts self
```

La respuesta es main, un término que se refiere al objeto que se crea al iniciar el intérprete.

self dentro de clases y módulos

En una clase o definición de módulo, self es la clase o el módulo al que pertenece el objeto:

```
class S
  puts 'Comenzó la clase S'
  puts self
  module M
    puts 'Módulo anidado S::M'
    puts self
  end
  puts 'De regreso en el nivel más superficial de S'
  puts self
end
```

La salida es:

```
Comenzó la clase S
S
Módulo anidado S::M
S::M
De regreso en el nivel más superficial de S
S
```

self dentro de los métodos

```
class S
  def m
    puts 'Clase S, metodo m:'
    puts self # <S:0x2835908>
  end
end
s = S.new
s.m
```

Duck Typing

A estas alturas, te habrás dado cuenta de que en Ruby no se declaran los tipos de variables o métodos: todo es un objeto. Los objetos en Ruby pueden ser modificados: siempre se pueden añadir métodos a posteriori. Por lo tanto, el comportamiento del objeto, puede alejarse de aquel suministrado por su clase.

En Ruby, nos fijamos menos en el tipo (o clase) de un objeto y más en sus capacidades. 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 llama "Duck Typing" porque está basado en el Test del Pato (Duck Test):

> Si camina como un pato, nada como un pato y hace "quack", podemos tratarlo como un pato. James Whitcomb Riley

Veamos el siguiente ejemplo:

```
# Comprobamos qué objetos responden al método t_str
puts ('Una cadena'.respond_to? :to_str)   # => true
puts (Exception.new.respond_to? :to_str)  # => true
puts (4.respond_to? :to_str) # => false
```

Este ejemplo, es una forma simple de la filosofía "pato typing": si un objeto hace quack como un pato (o actúa como un string), pues trátalo como un pato (o una cadena). Siempre hay que tratar a los objetos por lo que pueden hacer, mejoer que hacerlo por las clases de las que proceden o los módulos que incluyen.

Las excepciones (Exceptions) son un tipo de string que tienen información extra asociada con ellas. Sin embargo, aunque ellas no son una subclase de `String`, pueden ser tratadas como tales.

¡Tratémoslos como patos!

```
class Pato
  def quack
    'Quack!'
  end

  def nadar
    'Paddle paddle paddle...'
  end
end

class Ganso
  def honk
    'Honk!' # onomatopia de un pato
  end
  def nadar
    'Splash splash splash...'
  end
end

class GrabadoraDePatos
  def quack
    play
  end

  def play
    'Quack!'
  end
end
```

```ruby
# En este método, la Grabadora
# se comporta como un Pato
def haz_quack(pato)
  pato.quack
end
puts haz_quack(Pato.new)
puts haz_quack(GrabadoraDePatos.new)

# Para este método, el Ganso
# se comporta como un Pato
def haz_nadar(pato)
  pato.nadar
end
puts haz_nadar(Pato.new)
puts haz_nadar(Ganso.new)
```

Azúcar Sintáctico

Algunos de los patrones de programación se repiten tanto que los lenguajes de programación incluyen formas sintácticas que son abreviaciones para estos patrones. El único objetivo de estas abreviaciones es brindar un mecanismo para escribir menos código. Es el azúcar sintático.

Por ejemplo, a la hora de cambiar un atributo:

```
class Perro
  def initialize(raza)
    @raza = raza
  end
  attr_reader :raza, :nombre   # lector
  # método modificador
  def set_nombre(nm)
    @nombre = nm
  end
end

pe = Perro.new('Doberman')
pe.set_nombre('Benzy')
puts pe.nombre
```

Ruby permite definir métodos que terminan en `=`

```
def nombre=(nm)
  @nombre = nm
end

# usando el nuevo método
nombre=('Benzy')
# los paréntesis son opcionales,
# si es un sólo argumento
nombre='Benzy'
```

si empleamos "este azúcar" en el ejemplo:

```
class Perro
  def initialize(raza)
    @raza = raza
  end
  attr_reader :raza, :nombre # lector

  # modificador
  def nombre=(nb)
    @nombre = nb
  end
end

pe = Perro.new('Doberman')
#pe.nombre=('Benzy')
pe.nombre = 'Benzy'
puts pe.nombre
```

El signo igual es una forma familiar de asignar un valor; además que nos ahorramos poner los paréntesis.

RAILS: el uso del signo igual, de forma similar a la vista, es común en Rails.

Test de unidades

El test de unidades es un método para testear el código en pequeños trozos.

¿Por qué?

- Significa que nunca tendrás el problema de crear un error mientras solucionas otro.
- Significa que no tendrás que ejecutar tu programa y jugar con él (lo que es lento) para arreglar los errores. El testeo de unidades es mucho más rápido que el "testeo manual".
- Conociendo cómo usar las unidades de test, abre el mundo al Desarrollo Guiado por Pruebas (Test Driven Development, TDD).

Requisitos

- Cargar la biblioteca `test/unit`
- Hacer que la clase a testear sea una subclase de `Test::Unit::TestCase`
- Escribir los métodos con el prefijo `test_`
- Afirmar (`assert`) las cosas que decidas que sean ciertas.
- Ejecutar los tests y corregir los errores hasta que desaparezcan.

```
require 'test/unit'

class MiPrimerTest < Test::Unit::TestCase
  def test_de_verdad
    assert true
  end
end
```

Cada afirmación, es un método heredado de la clase `Test::Unit::TestCase:` Hay que echar un ojo al listado de las posibles afirmaciones (asserts) que podemos comprobar.

Ejemplo

Supongamos que queremos escribir una clase sencilla, Mates, que implemente operaciones aritméticas básicas. Queremos hacer distintos tests para comprobar que la suma, la resta, el producto y la división funcionan.

```
require 'mates'
require 'test/unit'

class TestDeMates < Test::Unit::TestCcase
  def test_suma
    assert_equal 4, Mates.run("2+2")
    assert_equal 4, Mates.run("1+3")
    assert_equal 5, Mates.run("5+0")
    assert_equal 0, Mates.run("-5 + 5")
  end

  def test_resta
    assert_equal 0, Mates.run("2-2")
    assert_equal 1, Mates.run("2-1")
    assert_equal -1, Mates.run("2-3")
  end
end
```

Si ejecutamos el programa, aparecerán siete puntos `.......`. Cada . es un test que se ha ejecutado, `E` es un error y cada `F` un fallo.

```
Started
.......
Finished in 0.015931 seconds.
7 tests, 13 assertions, 0 failures, 0 errors
```

Unidades de test negativas

Además de los tests positivos, también se pueden escribir unidades de tests negativas intentando romper el código. Esto puede incluir el testeo para excepciones que surgan de usar entradas como `Mates.run("a + 2")` o `Mates.run("4/0")`.

```
  def test_para_no_numericos
    assert_raises(ErrorNoNumerico) do
      Mates.run("a + 2")
    end
  end

  def test_division_por_cero
    assert_raises(ErrorDivisionPorZero) do
      Mates.run("4/0")
    end
  end
```

Automatizando tests: setup, teardown y rake

Algunas veces necesitamos que ocurran cosas antes y después de cada test. Los métodos setup y teardown son tus compañeros en esta aventura. Cualquier código escrito en setup será ejecutado antes del código, y el código escrito en teardown será ejecutado a posteriori.

Si estás escibiendo tests para todo tu código (como debería ser), el número de ficheros a testear empieza a crecer. Una cosa que puede facilitarte la vida, es automatizar los tests, y `rake` es la herramienta para este trabajo.

fichero_rake

```
require 'rake'
require 'rake/testtask'

task :default => [:test_units]

desc "Ejecutando los tests"
Rake::TestTask.new("test_units") { |t|
  t.pattern = 'test/*_test.rb'  # busca los ficheros acabados en '_test.rb'
  t.verbose = true
  t.warning = true
}
```

Básicamente, un fichero_rake define las tareas que `rake` puede hacer. En el fichero_rake, la tarea por defecto (la que sucede cuando se ejecuta rake en un directorio con un fichero_rake en él) es configurada hacia la tarea `tests_units`. En la tarea `tests_units`, rake es configurado para buscar ficheros en el directorio que terminen en `_test.rb` y los ejecute. Resumiendo: puedes poner todos los tests en un directorio y dejar que rake haga el trabajo.