Guía del usuario de Ruby

matz

Guía del usuario de Ruby

por matz

Tabla de contenidos

Lista de tablas

Contenidos

Ruby es “un lenguaje de programación orientado a objetos sencillo”. Al principio puede parecer un poco extraño, pero se ha diseñado para que sea fácil de leer y escribir. Esta *Guía del usuario de Ruby* permite ejecutar y utilizar Ruby y proporciona una visión de la naturaleza de Ruby que no se puede obtener del manual de referencia.

Capítulo 1. ¿Qué es Ruby?

Ruby es un “lenguaje de guiones (scripts) para una programación orientada a objetos rápida y sencilla”. ¿Qué significa esto?

Lenguaje de guiones interpretado:

- Posibilidad de realizar directamente llamadas al sistema operativo
- Potentes operaciones sobre cadenas de caracteres y expresiones regulares
- Retroalimentación inmediata durante el proceso de desarrollo

Rápido y sencillo:

- Son innecesarias las declaraciones de variables
- Las variables no tienen tipo
- La sintaxis es simple y consistente
- La gestión de la memoria es automática

Programación orientada a objetos:

- Todo es un objeto
- Clases, herencia, métodos, ...
- Métodos singleton
- Mixins por módulos
- Iteradores y cierres

También:

- Enteros de precisión múltiple
- Modelo de procesamiento de excepciones
- Carga dinámica
- Hilos

Si no estás familiarizado con alguno de los términos anteriores, continúa leyendo y no te preocupes. El mantra de Ruby es *Rápido y Sencillo* .

Capítulo 2. Inicio

Inicialmente hay que comprobar si se tiene instalado Ruby. Desde la línea de petición de comandos de la shell (aquí la representaremos por “%”, por lo tanto no introducir el % de los ejemplos), tecleamos:

```
% ruby -v
```

(**-v** le indica al intérprete que imprima la versión de Ruby), a continuación pulsamos la tecla Enter. Si está instalado Ruby, aparecerá el siguiente mensaje o algo similar:

```
% ruby -v
ruby 1.6.3 (2001-11-23) [i586-linux]
```

Si no está instalado, pide a tu administrador que lo instale, o hazlo tú mismo dado que Ruby es software libre sin restricciones de instalación o uso.

Comencemos ahora a jugar con Ruby. Se puede introducir directamente en la línea de comandos un programa Ruby utilizando la opción **-e**:

```
% ruby -e ’print "hola mundo\n"’
hola mundo
```

Un programa Ruby se puede almacenar en un fichero, lo que es mucho más adecuado.

```
% cat > test.rb
print "hola mundo\n"
^D
% cat test.rb
print "hola mundo\n"
%ruby test.rb
hola mundo
```

^D es *control-D*. Lo anterior es válido para UNIX. Si se está utilizando DOS, prueba con:

```
C:\ruby> copy con: test.rb
print "hola mundo\n"
^Z
C:\ruby> type test.rb
print "hola mundo\n"
c:\ruby> ruby test.rb
hola mundo
```

Al escribir código con más fundamento que éste, ¡se puede utilizar cualquier editor!.

Algunas cosas sorprendentemente complejas y útiles se pueden hacer con programas miniatura que caben en la línea de comandos. Por ejemplo, el siguiente programa reemplaza la cadena **foo** por **bar** en todos los ficheros cabecera y fuentes C del directorio de trabajo, realizando una copia de seguridad del fichero original a la que añade “.bak”

```
% ruby -i .bak -pe ’sub "foo", "bar"’ *.[ch]
```

El siguiente programa funciona como el comando **cat** de UNIX (aunque es más lento):

```
% ruby -pe 0 file
```

Capítulo 3. Ejemplos sencillos

Escribamos una función que calcula factoriales. La definición matemática de factorial es la siguiente:

(n==0) n! = 1

(sino) n! = n * (n-1)!

En Ruby se puede escribir así:

```
def fact(n)
  if n == 0
    1
  else
    n * fact(n-1)
  end
end
```

Se puede apreciar la aparición repetida de **end**. Debido a esto a Ruby se le conoce como un lenguaje “tipo Algol”. (Realmente la sintaxis de Ruby reproduce con más exactitud la del lenguaje Eiffel). También se puede apreciar la falta de la sentencia **return**. Es innecesaria debido a que una función Ruby devuelve lo último que haya evaluado. La utilización de **return** es factible aunque innecesaria.

Probemos la función factorial. Añadiendo una línea de código obtenemos un programa funcional:

```
# Programa para hallar el factorial de un número
# Guarda este programa como fact.rb

def fact(n)
  if n == 0
    1
  else
    n * fact(n-1)
  end
end

print fact(ARGV[0].to_i), "\n"
```

Aquí, **ARGV** es un array que contiene los parámetros de la línea de comandos y **to_i** convierte una cadena de caracteres a un entero.

```
% ruby fact.rb 1
1
% ruby fact.rb 5
120
```

¿Funcionaría con un parámetro igual a 40? Este valor podría provocar un desbordamiento en una calculadora...

```
% ruby fact.rb 40
815915283247897734345611269596115894272000000000
```

Funciona. Además Ruby puede tratar cualquier entero que quepa en la memoria del ordenador. ¡Por lo que se puede calcular el factorial de 400!:

```
% ruby fact.rb 400
64034522846623895262347970319503005850702583026002959458684445942802397169186831 4
36278478647463264676294350575035856810848298162883517435228961988646802997937341 6
54150838162426461942352307046244325015114448670890662773914918117331955996440709 5
49671345290477020322434911210797593280795101545372667251627877890009349763765710 3
26350331533965349868386831339352024373788157786791506311858702618270169819740062 9
83025308591298346162272304558339520759611505302236086810433297255194852674432232 4
38669948422404232599805551610635942376961399231917134063858996537970147827206606 3
20217379472010321356624613809077942304597360699567595836096158715129913822286578 5
79549361617654480453222007825818400848436415591229454275384803558374518022675900 0
61399560145595206127211192918105032491008000000000000000000000000000000000000000 0
00000000000000000000000000000000000000000000000000000000000000000000000000000000
```

No podemos verificar su validez a simple vista, pero lo debe ser ;-).

3.1. El bucle de entrada/evaluación

Al ejecutar Ruby sin parámetros, éste lee de la entrada estándar comandos y los ejecuta después de dar por finalizada la entrada:

```
% ruby
print "hola mundo\n"
print "adiós mundo\n"
^D
hola mundo
adiós mundo
```

Ruby incluye un programa llamado **eval.rb** que permite la introducción de código desde el teclado a través de un bucle iterativo que muestra los resultados a medida que se obtienen. Se utilizará ampliamente a lo largo del tutorial.

Si se tiene un terminal ANSI (esto será casi seguro si se está ejecutando alguna versión de UNIX; bajo DOS se debe instalar **ANSI.SYS** o **ANSI.COM**) se debería utilizar este **eval.rb** mejorado que permite autoindentación, informes sobre incidencias y resaltado por color. Si no, en el directorio **sample** de la distribución Ruby existe una versión no ANSI que funciona con cualquier terminal. A continuación se muestra una pequeña sesión con **eval.rb**:

```
%ruby eval.rb
ruby> print "hola mundo\n"
hola mundo.
  nil
ruby> exit
```

print produce *hola mundo*. La siguiente línea, es este caso *nil* informa sobre lo último que se ha evaluado; Ruby no distingue entre *sentencias* y *expresiones*, por lo tanto la evaluación de una pieza de código significa básicamente lo mismo que ejecutarla. Aquí *nil*, indica que *print* no devuelve ningún valor significativo. Obsérvese que se puede salir del bucle de interpretación con *exit*, aunque también funciona *^D*.

A lo largo de toda esta introducción **ruby>** representa la petición de entrada del pequeño pero útil programa **eval.rb**.

Capítulo 4. Cadenas

Ruby maneja tanto cadenas como datos numéricos. Las cadenas pueden estar entre comillas dobles("...") o comillas simples (’...’).

```
ruby> "abc"
"abc"
ruby> ’abc’
"abc"
```

Las comillas simples y dobles a veces tienen efectos diferentes. Una cadena de comillas dobles permite la presencia embebida de caracteres de escape precedidos por un backslash y la expresión de evaluación #{ }. Una cadena de comillas simples no realiza esta evaluación, lo que se ve es lo que se obtiene. Ejemplos:

```
ruby> print "a\nb\nc","\n"
a
b
c
nil
ruby> print ’a\nb\nc’,"\n"
a\nb\nc
nil
ruby> "\n"
"\n"
ruby> ’\n’
"\\n"
ruby> "\001"
"\001"
ruby> ’\001’
"\\001"
ruby> "abcd #{5*3} efg"
"abcd 15 efg"
ruby> var = " abc "
" abc "
ruby> "1234#{var}5678"
"1234 abc 5678"
```

El manejo de las cadenas en Ruby es más inteligente e intuitivo que en C. Por ejemplo, se pueden concatenar cadenas con + y se puede repetir una cadena varias veces con *:

```
ruby> "foo" + "bar"
"foobar"
ruby> "foo" * 2
"foofoo"
```

La concatenación de cadenas en C es más incómoda debido a la necesidad de una gestión explícita de la memoria:

```
char *s = malloc(strlen(s1)+strlen(s2) +1);
strcpy(s, s1);
```

```
strcat(s, s2);
/* ... */
free(s);
```

Ya que al usar Ruby no tenemos que considerar el espacio que va a ocupar una cadena, estamos liberados de la gestión de la memoria.

A continuación se muestran algunas cosas que se pueden hacer con las cadenas.

Concatenación:

```
ruby> word = "fo" + "o"
"foo"
```

Repetición:

```
ruby> word = word * 2
"foofoo"
```

Extracción de caracteres (obsérvese que los caracteres en Ruby son enteros):

```
ruby> word[0]
102             # 102 es el código ASCII de 'f'
ruby> word[-1]
111             # 111 es el código ASCII de 'o'
```

(Los índices negativos indican desplazamientos desde el final de la cadena, en vez del comienzo).

Extracción de subcadenas:

```
ruby> herb = "parsley"
"parsley"
ruby> herb[0,1]
"p"
ruby> herb[-2,2]
"ey"
ruby> herb[0..3]
"pars"
ruby> herb[-5..-2]
"rsle"
```

Verificación de la igualdad:

```
ruby> "foo" == "foo"
true
ruby> "foo" == "bar"
false
```

Pongamos ahora alguna de estas características en uso. Este acertijo es "Adivina la palabra", aunque tal vez la palabra "acertijo" es demasiado elevada para lo que viene ;-).

```
# Salvar como guess.rb
words = [’foobar’, ’baz’, ’quux’]
secret = words[rand(3)]

print "adivina? "
while guess = STDIN.gets
  guess.chop!
  if guess == secret
    print "¡Ganas!\n"
    break
  else
    print "Lo siento. Pierdes\n"
  end
  print "adivina? "
end
print "La palabra era ", secret, ".\n"
```

Por el momento no te preocupes de los detalles del código. A continuación se muestra una ejecución del programa del acertijo.

```
$ ruby guess.rb
adivina? foobar
Lo siento. Pierdes.
adivina? quux
Lo siento. Pierdes.
adivina? ^D
La palabra era baz.
```

(Debería haberlo hecho mejor dada la probabilidad de 1/3 de acertar).

Capítulo 5. Expresiones regulares

Realicemos un programa mucho más interesante. Es el momento de comprobar si una cadena satisface una descripción, que llamaremos *patrón*.

En estos patrones existen algunos caracteres y combinaciones de caracteres que tienen un significado especial, y son:

Tabla 5-1. Caracteres especiales en expresiones regulares

Símbolo	Descripción>
[]	Especificación de rango. (p.e. [a-z] representa una letra en el rango de la *a* a la *z*
\w	Letra o dígito; es lo mismo que [0-9A-Za-z]
\W	Ni letra, ni dígito
\s	Espacio, es lo mismo que [\t\n\r\f]
\S	No espacio
\d	Dígito; es lo mismo que [0-9]
\D	No dígito
\b	Backspace (0x08) (sólo si aparece en una especificación de rango)
\b	Límite de palabra (sólo si no aparece en una especificación de rango)
\B	No límite de palabra
*	Cero o más repeticiones de lo que precede
+	Una o más repeticiones de lo que precede
[m,n]	Al menos *m* y como máximo *n* de lo que precede
?	Al menos una repetición de lo que precede; es lo mismo que [0,1]
|	Puede coincidir con lo que precede o con lo que sigue
()	Agrupamiento

El término común para esto patrones que utilizan este extraño vocabulario es *expresión regular*. En Ruby, como en Perl, normalmente están rodeadas por barras inclinadas en vez de por comillas dobles. Si nunca antes se ha trabajado con expresiones regulares, es probable que parezcan cualquier cosa excepto *regulares*, pero sería inteligente dedicar algún tiempo a familiarizarse con ellas. Tienen un poder expresivo en su concisión que puede evitar muchos dolores de cabeza (y muchas líneas de código) si se necesita realizar coincidencia de patrones o cualquier otro tipo de manipulación con cadenas de texto.

Por ejemplo, supongamos que queremos comprobar si una cadena se ajusta a esta descripción: “Comienza con una f minúscula, a la que sigue exactamente una letra mayúscula y opcionalmente cualquier cosa detrás de ésta, siempre y cuando no haya más letras minúsculas.” Si se es un experimentado programador en C probablemente se haya escrito este tipo de código docenas de veces, ¿verdad? Admitádmoslo, es difícil mejorarlo. Pero en Ruby sólamente es necesario solicitar que se verifique la cadena contra la siguiente expresión regular **/^f[A-Z][^a-z]*$/**.

Y que decir de “¿Contiene la cadena un número hexadecimal entre ángulos?” No hay problema.

```
ruby> def chab(s) # contiene la cadena un hexadecinal entre ángulos
ruby|   (s =~ /<0[Xx][\dA-Fa-f]+>/) != nil
ruby| end
nil
```

```
ruby> chab "Este no es"
false
ruby> chab "¿Puede ser esta? (0x35)" # entre paréntesis, no ángulos
false
ruby> chab "¿O esta? <0x38z7e>"  # letra errónea
false
ruby> chab "OK esta si; <0xfc0004>"
true
```

Aunque inicialmente las expresiones regulares pueden parecer enigmáticas, se gana rápidamente satisfacción al ser capaz de expresarse con tanta economía.

A continuación se presenta un pequeño programa que nos permitirá experimentar con las expresiones regulares. Almacenémoslo como **regx.rb**, se ejecuta introduciendo en la línea de comandos **ruby regx.rb**

```
# necesita un terminal ANSI!!!

st = "\033[7m"
en = "\033[m"

while TRUE
  print "str> "
  STDOUT.flush
  str = gets
  break if not str
  str.chop!
  print "pat> "
  STDOUT.flush
  re = gets
  break if not re
  re.chop!
  str.gsub! re, "#{st}\\&#{en}"
  print str, "\n"
end
print "\n"
```

El programa necesita dos entradas, una con la cadena y otra con la expresión regular. La cadena se comprueba contra la expresión regular, a continuación muestra todas las partes de la cadena que coindicen con el patrón en vídeo inverso. No nos preocupemos de los detalles; analizaremos el código posteriormente.

```
str> foobar
pat> ^fo+
foobar
~~~
```

Lo resaltado es lo que aparecerá en vídeo inverso como resultado de la ejecución del programa. La cadena ’~~~’ es en beneficio de aquellos que usen visualizadores en modo texto.

Probemos algunos ejemplos más.

```
str> abc012dbcd555
pat> \d
abc012dbcd555
```

```
   ~~~     ~~~
```

Sorprendentemente y como indica la tabla al principio de este capítulo: **\d** no tiene nada que ver el carácter *d*, sino que realiza la coincidencia con un dígito.

¿Qué pasa si hay más de una forma de realizar la coincidencia con el patrón?.

```
str> foozboozer
pat> f.*z
foozboozer
~~~~~~~~
```

se obtiene *foozbooz* en vez de *fooz* porque las expresiones regulares tratan de obtener la coincidencia más larga posible.

A continuación se muestra un patrón para aislar la hora de un campo limitada por dos puntos.

```
str> WedFeb  7 08:58:04 JST 2001
pat> [0-9]+:[0-9]+(:[0-9]+)?
WedFeb  7 08:58:04 JST 2001
          ~~~~~~~~
```

=~ es el operador de coincidencia con expresiones regulares; devuelve la posición en la cadena donde se ha producido una coincidencia o **nil** si no la hay.

```
ruby> "abcdef" =~ /d/
3
ruby> "aaaaaa" =~ /d/
nil
```

Capítulo 6. Arrays

Se pueden crear un *array* listando elementos entre corchetes ([ ]) y separándolos por comas. Los arrays en Ruby pueden almacenar objetos de diferentes tipos.

```
ruby> ary = [1, 2, "3"]
[1, 2, "3"]
```

Los arrays se pueden concatenar y repetir, igual que las cadenas.

```
ruby> ary + ["foo", "bar"]
[1, 2, "3", "foo", "bar"]
ruby> ary * 2
[1, 2, "3", 1, 2, "3"]
```

Se pueden utilizar índices numéricos para acceder a cualquier parte del array.

```
ruby> ary[0]
1
ruby> ary[0,2]
[1, 2]
ruby> ary[-2]
2
ruby> ary[-2,2]
[2, "3"]
ruby> ary[-2..-1]
[2, "3"]
```

(Los índices negativos indican que se empieza a contar desde el final del array, en vez del principio).

Los arrays se pueden convertir a y obtener de cadenas utilizado **join** y **split** respectivamente:

```
ruby> str = ary.join(':')
"1:2:3"
ruby> str.split(':')
["1", "2", "3"]
```

Capítulo 7. Hashes

Un array asociativo contiene elementos que se pueden acceder, no a través de índices numéricos secuenciales, sino a través de *claves* que pueden tener cualquier tipo de valor. Estos arrays se conocen a veces como *hash* o *diccionario*; en el mundo Ruby se prefiere el término *hash*. Los hash se pueden crear mediante pares de elementos dentro de llaves ({ }). Se usa la clave para encontrar algo en un hash de la misma forma que se utiliza el índice para encontrar algo en un array.

```
ruby> h = {1 => 2, "2" => "4"}
{"2"=>"4", 1=>2}
ruby> h[1]
2
ruby> h["2"]
"4"
ruby> h[5]
nil
ruby> h[5] = 10  # añadimos un valor
10
ruby> h
{5=>10, "2"=>"4", 1=>2}
ruby> h[1]=nil # borramos un valor
nil
ruby> h[1]
nil
ruby> h
{5=>10, "2"=>"4", 1=>nil}
```

Capítulo 8. Retomando los ejemplos sencillos

Vamos ahora a desmontar el código de nuestros anteriores programas ejemplo. Para que sirva de referencia vamos a numerar las líneas de todos los guiones.

8.1. Factoriales

El siguiente guión aparece en el capítulo *Ejemplos sencillos*.

```
01 def fact(n)
02 if n == 0
03  1
04 else
05  n * fact(n-1)
06 end
07 end
08 print fact(ARGV[0].to_i), "\n"
```

Debido a que es la primera explicación de un código, vamos a ir línea por línea.

```
01 def fact(n)
```

En la primera línea, **def** es la sentencia que define una función (o con mayor precisión, un *método*; trataremos con más detalle los métodos en un capítulo posterior). Aquí se indica que la función **fact** toma un único argumento, que se llama **n**.

```
02 if n == 0
```

Con **if** comprobamos una condición. Si la condición es cierta, se evalúa la siguiente línea; si no independientemente de lo que siga se evalúa el **else**

```
03  1
```

Si la condición es cierta el valor del **if** es 1.

```
04 else
```

Si la condición no es cierta, se evalúa el código desde esta línea hasta el **end**.

```
05  n * fact(n - 1)
```

Si no se satisface la condición el valor de **if** es el resultado de multiplicar **fact(n-1)** por **n**.

```
06 end
```

El primer **end** cierra la sentencia **if**.

```
07 end
```

El segundo **end** cierra la sentencia **def**.

```
08 print fact(ARGV[0].to_i), "\n"
```

Llama a la función **fact()** con el argumento especificado en la línea de comandos, e imprime el resultado.

ARGV es un array que contiene los argumentos de la línea de comandos. Los miembros de **ARGV** son cadenas por lo que hay que convertirlos a números enteros con **to_i**. Ruby no convierte automáticamente las cadenas a números como hace Perl.

Hmmm... ¿Qué pasa si alimentamos este programa con un número negativo? ¿Se ve cuál es el problema? ¿Cómo se podría solucionar?

8.2. Cadenas

A continuación examinaremos el programa acertijo del capítulo sobre las cadenas

```
01 words = ['foobar', 'baz', 'quux']
02 secret = words[rand(3)]
03
04 print "adivina? "
05 while guess = STDIN.gets
06  guess.chop!
07  if guess == secret
08    print "¡Ganas!\n"
09    break
10  else
11    print "Lo siento. Pierdes\n"
12  end
13  print "adivina? "
14 end
15 print "La palabra era ", secret, ".\n"
```

En este programa se utiliza una nueva estructura de control, **while**. El código entre el **while** y su correspondiente **end** se ejecutará repetidamente mientras la condición especificada se mantenga cierta.

rand(3) de la línea número 2 devuelve un número aleatorio dentro del rango de 0 a 2. Este número se utiliza para extraer uno de los elementos del array **words**.

En la línea 5 se lee una línea de la entrada estándar con el método **STDIN.gets**. Si aparece el fin del fichero (EOF - End Of File), **gets** devuelve **nil**. Por lo tanto el código asociado con el **while** se repetirá hasta encontrar un ^D (o ^Z bajo DOS), que representa el fin de fichero

En la línea 6 **guess.chop!** elimina el último carácter de **guess**; en este caso siempre el carácter de línea nueva.

En la línea 15 se imprime la palabra secreta. Se ha escrito como una sentencia print con tres argumentos (que se imprimen uno detrás del otro), pero hubiera tenido la misma efectividad el hacerlo con un único argumento escribiendo **secret** como **#{secret}** para resaltar que la variable se debe evaluar y no imprimir la palabra literal:

```
print "la palabra era #{secret}. \n"
```

8.3. Expresiones regulares

Por último examinaremos el programa del capítulo sobre expresiones regulares.

```
01 st = "\033[7m"
02 en = "\033[m"
03
04 while TRUE
05  print "str> "
06  STDOUT.flush
07  str = gets
08  break if not str
09  str.chop!
10  print "pat> "
11  STDOUT.flush
12  re = gets
13  break if not re
14  re.chop!
15  str.gsub! re, "#{st}\\&#{en}"
16  print str, "\n"
17 end
18 print "\n"
```

En la línea 4, se ha fijado la condición del **while** a **true** con lo que se obtiene un bucle infinito. Sin embargo se han colocado sentencias **break** en las líneas octava y decimotercera para salir del bucle. Estos dos **breaks** ejemplifican también el uso de los modificadores **if**. Un "modificador **if**" ejecuta la sentencia del lado izquierdo si y sólo si se satisface la condición especificada.

Hay más cosas que decir sobre **chop!** (veanse las líneas 9 y 14). En Ruby se añade, por convención, *!* o *?* al final de ciertos nombre de métodos. El marca de exclamación (*!*, pronunciada como un "bang!" sonoro) recalca algo potencialmente peligroso, es decir, algo que puede modificar el valor de lo que toca. **chop!** afecta directamente a la cadena pero **chop** sin el signo de exclamación actúa sobre una copia. A continuación se muestra la diferencia.

```
ruby> s1 = "forth"
"forth"
ruby> s1.chop!     # modifica s1
"fort"
ruby> s2 = s1.chop # sitúa en s2 una copia de la modificación
"for"
ruby> s1           # ... sin afectar a  s1
"fort"
```

Posteriormente no encontraremos con nombres de métodos que finalizan con un signo de interrogación (*?*, pronunciada a veces como un "huh?" sonoro). Esto indica que el método es un "predicado", aquel que puede devolver o **true** o **false**.

La línea 15 requiere una especial atención. Primero, se observa que **gsub!** es otro de los denominados métodos destructivos. Modifica **str** al reemplazar toda coincidencia del patrón **re** (**sub** significa sustituir, la **g** inicial indica que la sustitución es global, es decir reemplaza todas las coincidencias que hay en la cadena no sólo la primera encontrada). Hasta el momento todo parece correcto pero, ¿Por qué reemplazamos las coincidencias del texto? **st** y **en** se definieron en las líneas 1 y 2 como secuencias ANSI que presentan el color del texto como invertido o normal respectivamente. En la línea 15 se encuentran encerradas entre **#{}** para asegurar que se interpreten por lo que son (y no se impriman

los *nombres* de las variables). Entre ella se ve "\\&". Esto es un pequeño truco. Dado que la sentencia de reemplazo se encuentra entre comillas dobles, los dos backslashes se interpretarán como uno solo, que **gsub!** verá como "\&" que no es otra cosa que el código que representa la primera coincidencia del patrón. Por lo tanto la nueva cadena, al imprimirse, será igual que la antigua, excepto que las partes que coinciden con el patrón aparecen resaltadas en vídeo inverso.

Capítulo 9. Estructuras de control

Este capítulo explora más sentencias de control de Ruby.

9.1. case

Se utiliza la sentencia **case** para comprobar una secuencia de condiciones. Superficialmente se parece al **switch** de C y Java pero es considerablemente más potente como veremos.

```
ruby> i=8
8
ruby> case i
ruby| when 1, 2..5
ruby|   print "1..5\n"
ruby| when 6..10
ruby|   print "6..10\n"
ruby| end
6..10
nil
```

2..5 es una expresión que representa un *rango* entre 2 y 5 inclusive. La siguiente expresión verifica si el valor **i** cae dentro del rango:

```
(2..5) === i
```

La sentencia **case** utiliza internamente el operador === para verificar las distintas condiciones. Dentro de la naturaleza orientada a objetos de Ruby, === lo interpreta el objeto que aparece en la condición **when**. Por ejemplo, el código que sigue comprueba en el primer **when** la igualdad de cadenas y en el segundo la coincidencia con una expresión regular.

```
ruby> case 'abcdef'
ruby| when 'aaa', 'bbb'
ruby|   print "aaa o bbb\n"
ruby| when /def/
ruby|   print "incluye /def/\n"
ruby| end
incluye /def/
nil
```

9.2. while

Ruby proporciona medios adecuados para la construcción de bucles, aunque veremos en el siguiente capítulo que si se aprende a utilizar los *iteradores* a menudo hace innecesario su utilización explícita.

Un **while** es un **if** repetido. Se ha utilizado en nuestros programas acertijo adivina-palabra y en las expresiones regulares (ver el capítulo anterior); allí tomaba la forma **while `condición` ... end** que rodeaba el código a repetir mientras la *condición* fuera cierta. Pero **while** e **if** se pueden aplicar fácilmente a sentencias individuales:

```
ruby> i = 0
0
ruby> print "Es cero.\n" if i == 0
Es cero.
nil
ruby> print "Es negativo\n" if i < 0
nil
ruby> print "#{i+=1}\n" while i < 3
1
2
3
nil
```

Algunas veces se necesita la condición de comprobación negada. Un **unless** es un **if** negado y un **until** es un **while** negado. Dejamos estas sentencias para que se experimente con ellas.

Existen cuatro formas de interrumpir el progreso de un bucle desde su interior. La primera **break**, como en C, sale completamente del bucle. La segunda **next** salta al principio de la siguiente iteración del bucle (se corresponde con la sentencia **continue** del C). La tercera **redo** reinicia la iteración en curso. A continuación se muestra un extracto de código en C que ilustra el significado de **break**, **next**, y **redo**:

```
while (condicion) {
label_redo:
goto label_next    /*next*/
goto label_break   /*break*/
goto label_redo    /*redo*/
;
;
label_next:
}
label_break:
;
```

La cuarta forma de salir del interior de un bucle es **return**. La evaluación de **return** provoca la salida no sólo del bucle sino también del método que contiene el bucle. Si se le pasa un argumento, lo devolverá como retorno de la llamada al método, si no el retorno será **nil**.

9.3. for

Los programadores en C se estarán preguntando cómo se construye un bucle **for**. En Ruby, el bucle **for** es mucho más interesante de lo que cabía esperar. El siguiente bucle se ejecuta una vez por cada elemento de la colección.

```
for elem in coleccion
  ...
end
```

La colección puede ser un rango de valores (esto es lo que la mayoría de la gente espera cuando se habla de bucles for):

```
ruby> for num in (4..6)
ruby|   print num, "\n"
ruby| end
4
5
6
4..6
```

Puede ser cualquier tipo de colección como por ejemplo un array:

```
ruby> for elm in [100,-9.6,"pickle"]
ruby|   print "#{elm}\t(#{elm.type})\n"
ruby| end
100     (Fixnum)
-9.6    (Float)
pickle   (String)
[100, -9.6, "pickle"]
```

Saliéndonos un poco del tema, **for** es realmente otra forma de escribir **each**, el cual es nuestro primer ejemplo de iterador. La siguientes dos estructuras son equivalentes:

```
# Si utilizas C o Java, puedes preferir esta estructura
for i in coleccion
  ..
end

# si utilizas Smalltalk, puedes preferir esta otra
coleccion.each {|i|
  ...
}
```

Con frecuencia se puede sustituir los bucles convencionales por iteradores y una vez acostumbrado a utilizarlos es generalmente más sencillo tratar con éstos. Por lo tanto, avancemos y aprendamos más sobre ellos.

Capítulo 10. Iteradores

Los iteradores no son un concepto original de Ruby. Son comunes en otros lenguajes orientados a objetos. También se utilizan en Lisp aunque no se les conoce como iteradores. Sin embargo este concepto de iterador es muy poco familiar para muchas personas por lo que se explorará con detalle.

Como ya se sabe, el verbo *iterar* significa hacer la misma cosa muchas veces, por lo tanto un *iterador* es algo que hace la misma cosa muchas veces.

Al escribir código se necesitan bucles en diferentes situaciones. En C, se codifican utilizando **for** o **while**. Por ejemplo:

```
char *str;
for (str = "abcdefg"; *str != '\0'; str++) {
  /* aquí procesamos los caracteres */
}
```

La sintaxis del **for(...)** de C nos dota de una abstracción que nos ayuda en la creación de un bucle pero, la comprobación de si ***str** es la cadena nula requiere que el programador conozca los detalles de la estructura interna de una cadena. Esto hace que C se parezca a un lenguaje de bajo nivel. Los lenguajes de alto nivel se caracterizan por un soporte más flexible a la iteración. Consideremos el siguiente guión de la shell **sh**:

```
#!/bin/sh

for i in *.[ch]; do
  # ... aquí se haría algo con cada uno de los ficheros
done
```

Se procesarían todos los ficheros fuentes en C y sus cabeceras del directorio actual, el comando de la shell se encargaría de los detalles de coger y sustituir los nombres de los ficheros uno por uno. Pensamos que este es un método de trabajo a nivel superior que C, ¿Verdad?

Pero hay más cosas a tener en cuenta: aunque está bien que un lenguaje tenga iteradores para todos los tipos de datos definidos en él, es decepcionante tener que volver a escribir bucles de bajo nivel para los tipos de datos propios. En la POO, los usuarios definen sus propios tipos de datos a partir de otros, por lo tanto, esto puede ser un problema serio.

Luego, todos los lenguajes OO incluyen ciertas facilidades de iteración. Algunos lenguajes proporcionan clases especiales con este propósito; Ruby nos permite definir directamente iteradores.

El tipo **strings** de Ruby tiene algunos iteradores útiles:

```
ruby> "abc".each_byte{|c| printf"{%c}", c}; print "\n"
{a}{b}{c}
nil
```

each_byte es un iterador sobre los caracteres de una cadena. Cada carácter se sustituye en la variable local **c**. Esto se puede traducir en algo más parecido a C ...

```
ruby> s="abc";i = 0
0
ruby> while i < s.length
```

```
ruby|   printf "{%c}",s[i]; i+=1
ruby| end; print "\n"
{a}{b}{c}
nil
```

... sin embargo el iterador **each_byte** es a la vez conceptualmente más simple y tiene más probabilidades de seguir funcionando correctamente incluso cuando, hipotéticamente, la clase **string** se modifique radicalmente en un futuro. Uno de los beneficios de los iteradores es que tienden a ser robustos frente a tales cambios, además, ésta es una característica del buen código en general. (Si, tengamos paciencia también hablaremos de lo que son las clases)

each_line es otro iterador de **String**.

```
ruby> "a\nb\nc\n".each_line{|l| print l}
a
b
c
"a\nb\nc\n"
```

Las tareas que más esfuerzo llevan en C (encontrar los delimitadores de línea, generar subcadenas, etc.) se evitan fácilmente utilizando iteradores.

La sentencia **for** que aparece en capítulos previos itera como lo hace el iterador **each**. El iterador **each** de **String** funciona de igual forma que **each_line**, reescribamos ahora el ejemplo anterior con un **for**:

```
ruby> for l in "a\nb\nc\n"
ruby|   print l
ruby| end
a
b
c
"a\nb\nc\n"
```

Se puede utilizar la sentencia de control **retry** junto con un bucle de iteración y se repetirá la iteración en curso desde el principio.

```
ruby> c = 0
0
ruby> for i in 0..4
ruby|   print i
ruby|   if i == 2 and c == 0
ruby|           c = 1
ruby|           print "\n"
ruby|           retry
ruby|   end
ruby| end; print "\n"
012
01234
nil
```

A veces aparece **yield** en la definición de un iterador. **yield** pasa el control al bloque de código que se pasa al iterador (esto se explorará con más detalle es el capítulo sobre los objetos procedimiento). El siguiente ejemplo define el iterador **repeat**, que repite el bloque de código el número de veces especificado en el argumento.

```
ruby> def repeat(num)
ruby|   while num > 0
ruby|           yield
ruby|           num -= 1
ruby|   end
ruby| end
nil
ruby> repeat(3){ print "foo\n" }
foo
foo
foo
nil
```

Con **retry** se puede definir un iterador que funciona igual que **while**, aunque es demasiado lento para ser práctico.

```
ruby> def WHILE(cond)
ruby|   return if not cond
ruby|   yield
ruby|   retry
ruby| end
nil
ruby> i=0;WHILE(i<3){ print i; i+=1 }
012nil
```

¿Se entiende lo que son los iteradores? Existen algunas restricciones pero se pueden escribir iteradores propios; y de hecho, al definir un nuevo tipo de datos, es conveniente definir iteradores adecuados para él. En este sentido los ejemplos anteriores no son terriblemente útiles. Volveremos a los iteradores cuando se sepa lo que son las clases.

Capítulo 11. Pensamiento orientado a objetos

La *orientación a objetos* es una palabra con gancho. Llamar a cualquier cosa "orientada a objetos" puede hacerla parecer más elegante. Ruby reclama ser un lenguaje de guiones orientado a objetos: pero, ¿Qué significa exactamente "orientado a objetos"?

Existe una gran variedad de respuestas a esta pregunta, y probablemente todas ellas se pueden reducir a la misma cosa. En vez de recapitular demasiado deprisa, pensemos un momento en el paradigma de la programación tradicional.

Tradicionalmente, un problema informático se ataca produciendo algún tipo de *representación de datos* y *procedimientos* que operan sobre esos datos. Bajo este modelo, los datos son inertes, pasivos e incapaces. Están a la completa merced de un gran cuerpo procedimental, que es activo, lógico y todopoderoso.

El problema con esta aproximación es, que los programas los escriben programadores, que son humanos, que sólo pueden retener cierto número de detalles en sus cabezas en un momento determinado. A medida que crece el proyecto, el núcleo procedimental crece hasta un punto que se hace difícil recordar cómo funciona todo el conjunto. Pequeños lapsos de pensamiento o errores tipográficos llegan a ser errores muy ocultos. Empiezan a surgir interacciones complejas e inintencionadas dentro de este núcleo y el mantenimiento se convierte en algo parecido a transportar un calamar gigante intentado que ninguno de sus tentáculos te alcance la cara. Existen políticas de programación que ayudan a minimizar y localizar errores dentro de este paradigma tradicional pero existe una solución mejor que pasa fundamentalmente por cambiar la forma de trabajar.

Lo que hace la programación orientada a objetos es, delegar la mayoría del trabajo mundano y repetitivo *a los propios datos*; modifica el concepto de los datos que pasan de *pasivos* a *activos*. Dicho de otra forma.

- Dejamos de tratar cada pieza de dato como una caja en la que se puede abrir su tapa y arrojar cosas en ella.
- Empezamos a tratar cada pieza de dato como una máquina funcional cerrada con unos pocos interruptores y diales bien definidos.

Lo que se define anteriormente como una "máquina" puede ser, en su interior, algo muy simple o muy complejo. No se puede saber desde el exterior y no se nos permite abrir la máquina (excepto cuando estamos completamente seguros de que algo está mal en su diseño), por lo que se nos obliga a conmutar interruptores y leer los diales para interactuar con los datos. Una vez construida, no queremos tener que pensar en como funciona internamente.

Se podría pensar que estamos haciendo más trabajo nosotros mismos, pero esta forma de trabajo tiende a ser un buen método para evitar que vayan mal todo tipo de cosas.

Comencemos con un ejemplo que es demasiado simple para tener algún valor práctico pero que al menos muestra parte del concepto. Nuestro coche consta de un odómetro. Su trabajo consiste en llevar un registro de la distancia recorrida desde la última vez que se pulsó el botón de reinicialización. ¿Cómo podríamos representar esto en un lenguaje de programación? En C, el odómetro sería, simplemente, una variable numérica de tipo **float**. El programa manipularía esa variable aumentando el valor en pequeños incrementos y ocasionalmente la reinicializaría a cero cuando fuese apropiado. ¿Qué hay de malo en esto? Un error en el programa podría asignar un valor falso a la variable, por cualquier número de razones inesperadas. Cualquiera que haya programado en C sabe que se pueden perder horas o días tratando de encontrar ese error que una vez encontrado parece absurdamente simple. (El momento de encontrar el error es indicado por una sonora palmada en la frente).

En un contexto orientado a objetos, el mismo problema se puede atacar desde un ángulo completamente diferente. La primera cosa que se pregunta un programador al diseñar el odómetro no es "¿qué tipos de datos son los más cercanos para representar esta cosa?" sino "¿cómo se supone que actúa esta cosa?". La diferencia termina siendo profunda. Es necesario dedicar cierto tiempo a decidir para qué es exactamente un odómetro y cómo se espera que el mundo

exterior interactúe con el. Se decide entonces construir una pequeña máquina con controles que permitan incrementar, reinicializar y leer su valor y nada más.

El odómetro se crea sin un mecanismo para asignarle un valor arbitrario, ¿Por qué? porque es de todos sabido que los odómetros no trabajan de esa forma. Existen sólo unas cuantas cosas que un odómetro puede hacer, y sólo permitimos esas cosas. Así, si alguien desde un programa trata de asignar algún otro valor (por ejemplo, la temperatura límite del sistema de control de climatización del vehículo) al odómetro, aparece de inmediato una indicación de lo que va mal. Al ejecutar el programa se nos dice (o posiblemente, al compilarlo dependiendo de la naturaleza del lenguaje) que *No se nos permite asignar valores arbitrarios al objeto Odómetro*. El mensaje podría ser menos preciso, pero sí razonablemente próximo al problema. Esto no evita el error, ¿verdad? pero apunta rápidamente en la dirección de la causa. Esta es sólo alguna de múltiples formas en las que la programación OO nos puede evitar muchas pérdidas de tiempo.

Existe, normalmente, un nivel de abstracción superior a éste porque resulta que es igual de fácil construir una factoría que hace máquinas como hacer una máquina individual. Es poco probable que construyamos un único odómetro, sino que nos preparamos para construir cualquier cantidad de odómetros a partir de un único patrón. El patrón (o si los prefieres, la factoría de odómetros) es lo que se conoce como *clase* y el odómetro individual sacado del patrón (o construido en la factoría) se conoce como *objeto*. La mayoría de los lenguajes OO necesitan una clase para tener un nuevo tipo de objeto pero Ruby no.

Conviene resaltar aquí que la utilización de un lenguaje OO no *obliga* a un diseño OO válido. Es posible, en cualquier lenguaje, escribir código poco claro, descuidado, mal concebido, erróneo e inestable. Lo que permite Ruby (en oposición, especialmente, a C++) es que la práctica de la programación OO sea lo suficientemente natural para que, incluso, trabajando a pequeña escala no se sienta la necesidad de recurrir a un código mal estructurado por evitar esfuerzo. Se tratará la forma en que Ruby logra este admirable propósito a medida que avancemos en esta guía; el próximo tema serán los “interruptores y diales” (métodos del objeto) y a partir de aquí pasaremos a las “factorías” (clases). ¿Sigues con nosotros?

Capítulo 12. Métodos

¿Qué es un método? En la programación OO no se piensa en operar sobre los datos directamente desde el exterior de un objeto; si no que los objetos tienen algún conocimiento de cómo se debe operar sobre ellos (cuando se les pide amablemente). Podríamos decir que se pasa un mensaje al objeto y este mensaje obtiene algún tipo de acción o respuesta significativa. Esto debe ocurrir sin que tengamos necesariamente algún tipo de conocimiento o nos importe como realiza el objeto, interiormente, el trabajo. Las tareas que podemos pedir que un objeto realice (o lo que es lo mismo, los mensajes que comprende) son los *métodos*.

En Ruby, se llama a un método con la notación punto (como en C++ o Java). El objeto con el que nos comunicamos se nombra a la izquierda del punto.

```
ruby> "abcdef".length
6
```

Intuitivamente, *a este objeto cadena se le está pidiendo que diga la longitud que tiene*. Técnicamente, se está llamando al método **length** del objeto "**abcdef**".

Otros objetos pueden hacer una interpretación un poco diferente de **length**. La decisión sobre cómo responder a un mensaje se hace al vuelo, durante la ejecución del programa, y la acción a tomar puede cambiar dependiendo de la variable a que se haga referencia.

```
ruby> foo = "abc"
"abc"
ruby> foo.length
3
ruby> foo = ["abcde","fghij"]
["abcde", "fghij"]
ruby> foo.length
2
```

Lo que indicamos con **length** puede variar dependiendo del objeto con el que nos comunicamos. En el primer ejemplo le pedimos a **foo** su longitud, como referencia a una cadena simple, sólo hay una respuesta posible. En el segundo ejemplo, **foo** referencia a un array, podríamos pensar que su longitud es 2, 5 ó 10; pero la respuesta más plausible es 2 (los otros tipos de longitud se podrían obtener si se desea)

```
ruby> foo[0].length
5
ruby> foo[0].length + foo[1].length
10
```

Lo que hay que tener en cuenta es que, el array *conoce lo que significa ser un array*. En Ruby, las piezas de datos llevan consigo ese conocimiento por lo que las solicitudes que se les hace se pueden satisfacer en las diferentes formas adecuadas. Esto libera al programador de la carga de memorizar una gran cantidad de nombres de funciones, ya que una cantidad relativamente pequeña de nombre de métodos, que corresponden a conceptos que sabemos como expresar en lenguaje natural, se pueden aplicar a diferentes tipos de datos siendo el resultado el que se espera. Esta característica de los lenguajes OO (que, IMHO[1], Java ha hecho un pobre trabajo en explotar), se conoce como *polimorfismo*

Cuando un objeto recibe un mensaje que no conoce, “salta” un error:

```
ruby> foo = 5
5
ruby> foo.length
ERR: (eval):1: undefined method ‘length’ for 5:Fixnum
```

Por lo tanto hay que conocer qué métodos son aceptable para un objeto, aunque no se necesita saber como son procesados.

Si se pasan argumentos a un método, éstos van normalmente entre paréntesis.

```
objeto.metodo(arg1, arg2)
```

pero se pueden omitir, si su ausencia no provoca ambigüedad

```
objeto.metodo arg1, arg2
```

En Ruby existe una variable especial **self** que referencia al objeto que llama a un método. Ocurre con tanta frecuencia que por conveniencia se omite en las llamadas de un método dentro de un objeto a sus propios métodos:

```
self.nombre_de_metodo(args ...)
```

es igual que:

```
nombre_de_metodo(args ...)
```

Lo que conocemos tradicionalmente como *llamadas a funciones* es esta forma abreviada de llamar a un método a través de **self**. Esto hace que a Ruby se le conozca como un lenguaje orientado a objetos puro. Aún así, los métodos funcionales se comportan de una forma muy parecida a las funciones de otros lenguajes de programación en beneficio de aquellos que no asimilen que las llamadas a funciones son realmente llamadas a métodos en Ruby. Se puede hablar de *funciones* como si no fuesen realmente métodos de objetos, si queremos.

Notas

1. In My Honest Opinion (En mi sincera opinión)

Capítulo 13. Clases

El mundo real está lleno de objetos que podemos clasificar. Por ejemplo, un niño muy pequeño es probable que diga “guau guau” cuando vea un perro, independientemente de su raza; naturalmente vemos el mundo en base a estas categorías.

En terminología OO, una categoría de objetos, como “perro”, se denomina *clase* y cualquier objeto determinado que pertenece a una clase se conoce como *instancia* de esa clase.

Generalmente, en Ruby y en cualquier otro lenguaje OO, se define primero las características de una clase, luego se crean las instancias. Para mostrar el proceso, definamos primero una clase muy simple **Perro**.

```
ruby> class Perro
ruby|   def ladra
ruby|           print "guau guau\n"
ruby|   end
ruby| end
nil
```

En Ruby, la definición de una clase es la región de código que se encuentra entre las palabras reservadas **class** y **end**. Dentro de esta área, **def** inicia la definición de un *método*, que como se dijo en el capítulo anterior, corresponde con algún comportamiento específico de los objetos de esa clase.

Ahora que tenemos definida la clase **Perro**, vamos a utilizarla:

```
ruby> rufi = Perro.new
#<Perro:0x401c444c>
```

Hemos creado una instancia nueva de la clase **Perro** y le hemos llamado **rufi**. El método **new** de cualquier clase, crea una nueva instancia. Dado que **rufi** es un **Perro**, según la definición de la clase, tiene las propiedades que se decidió que un **Perro** debía tener. Dado que la idea de *Perrunidad* es muy simple, sólo hay una cosa que puede hacer **rufi**

```
ruby> rufi.ladra
guau guau
nil
```

La creación de una instancia de una clase se conoce, a veces, como *instanciación*. Es necesario tener un perro antes de experimentar el placer de su conversación; no se puede pedir simplemente a la clase **Perro** que ladre para nosotros:

```
ruby> Perro.ladra
ERR: (eval):1: undefined method ‘ladra’ for Perro:Class
```

Tiene el mismo sentido que intentar *comer el concepto de un sándwich*

Por otro lado, si queremos oír el sonido de un perro sin estar emocionalmente atados, podemos crear (instanciar) un perro efímero, temporal y obtener un pequeño sonido antes de que desaparezca.

```
ruby> (Perro.new).ladra  # o también,  Perro.new.ladra
guau guau
nil
```

Pero un momento, "¿qué es todo esto de que a continuación el pobre tipo desaparece?". Pues es verdad, si no nos preocupamos de darle un nombre (como hicimos con **rufi**) el recolector de basura automático de Ruby decide que se trata de un perro perdido, no deseado y sin piedad se deshace de él. Ciertamente está muy bien, porque podemos crear todos los perros que queramos.

Capítulo 14. Herencia

La clasificación de los objetos en nuestra vida diaria es evidentemente jerárquica. Sabemos que todos los *gatos son mamíferos* y que *todos los mamíferos son animales*. Las clases inferiores *heredan* características de las clases superiores a las que pertenecen. Si todos los mamíferos respiran, entonces los gatos respiran.

Este concepto se puede expresar en Ruby:

```
ruby> class Mamifero
ruby|   def respira
ruby|           print "inhalar y exhalar\n"
ruby|   end
ruby| end
nil
ruby> class Gato<Mamifero
ruby|   def maulla
ruby|           print "miau \n"
ruby|   end
ruby| end
nil
```

Aunque no se dice cómo respira un **Gato**, todo gato heredará ese comportamiento de **Mamifero** dado que se ha definido **Gato** como una subclase de **Mamifero**. En terminología OO, la clase inferior es una *subclase* de la clase superior que es una *superclase*. Por lo tanto, desde el punto de vista del programador, los gatos obtienen gratuitamente la capacidad de respirar; a continuación se añade el método **maulla**, así nuestro gato puede respirar y maullar.

```
ruby> tama = Gato.new
#<Gato:0x401c41b8>
ruby> tama.respira
inhalar y exhalar
nil
ruby> tama.maulla
miau
nil
```

Existen situaciones donde ciertas propiedades de las superclases no deben heredarse por una determinada subclase. Aunque en general los pájaros vuelan, los pingüinos es una subclase de los pájaros que no vuelan.

```
ruby> class Pajaro
ruby|   def aseo
ruby|           print "me estoy limpiando las plumas."
ruby|   end
ruby|   def vuela
ruby|           print "estoy volando."
ruby|   end
ruby| end
nil
ruby> class Pinguino<Pajaro
ruby|   def vuela
ruby|           fail "Lo siento. yo sólo nado."
```

```
ruby|   end
ruby| end
nil
```

En vez de definir exhaustivamente todas las características de cada nueva clase, lo que se necesita es añadir o redefinir las diferencias entre cada subclase y superclase. Esta utilización de la herencia se conoce como *programación diferencial*. Y es uno de los beneficios de la programación orientada a objetos.

Capítulo 15. Redefinición de métodos

En una subclase se puede modificar el comportamiento de las instancias redefiniendo los métodos de la superclase.

```
ruby> class Humano
ruby|   def identidad
ruby|           print "soy una persona.\n"
ruby|   end
ruby|   def tarifa_tren(edad)
ruby|           if edad < 12
ruby|                   print "tarifa reducida.\n"
ruby|           else
ruby|                   print "tarifa normal. \n"
ruby|           end
ruby|   end
ruby| end
nil
ruby> Humano.new.identidad
soy una persona.
nil
ruby> class Estudiante<Humano
ruby|   def identidad
ruby|           print "soy un estudiante.\n"
ruby|   end
ruby| end
nil
ruby> Estudiante.new.identidad
soy un estudiante.
nil
```

Supongamos que en vez de reemplazar el método **identidad** lo que queremos es mejorarlo. Para ello podemos utilizar **super**

```
ruby> class Estudiante2<Humano
ruby|   def identidad
ruby|           super
ruby|           print "también soy un estudiante.\n"
ruby|   end
ruby| end
nil
ruby> Estudiante2.new.identidad
soy una persona.
también soy un estudiante.
nil
```

super nos permite pasar argumentos al método original. Se dice que hay dos tipos de personas ...

```
ruby> class Deshonesta<Humano
ruby|   def tarifa_tren(edad)
ruby|           super(11)       #quiero una tarifa barata
```

```
ruby|   end
ruby| end
nil
ruby> Deshonesta.new.tarifa_tren(25)
tarifa reducida.
nil
ruby> class Honesta<Humano
ruby|   def tarifa_tren(edad)
ruby|           super(edad)     #pasa el argumento entregado
ruby|   end
ruby| end
nil
ruby> Honesta.new.tarifa_tren(25)
tarifa normal.
nil
```

Capítulo 16. Control de accesos

Se ha dicho anteriormente que Ruby no tiene funciones, sólo métodos. Sin embargo existe más de una clase de métodos. En esta capítulo vamos a presentar el *control de accesos*.

Vamos a considerar lo que pasa cuando se define un método en el “nivel superior”, no dentro de una clase. Se puede pensar que dicho método es análogo a una *función* de un lenguaje más tradicional como C.

```
ruby> def square(n)
ruby|   n * n
ruby| end
nil
ruby> square(5)
25
```

Nuestro nuevo método parece que no pertenece a ninguna clase, pero de hecho Ruby se lo asigna a la clase **Object**, que es la superclase de cualquier otra clase. Como resultado de esto cualquier objeto es capaz de utilizar este método. Esto es cierto, pero existe un pequeño pero; es un método *privado* a cada clase. A continuación hablaremos más de lo que esto significa, pero una de sus consecuencias es que sólo se puede llamar de la siguiente forma

```
ruby> class Foo
ruby|   def fourth_power_of (x)
ruby|           square(x) * square(x)
ruby|   end
ruby| end
nil
ruby> Foo.new.fourth_power_of 10
10000
```

No se nos permite aplicar explícitamente el método a un objeto:

```
"fish".square(5)
ERR: (eval):1: private method ‘square’ called for "fish":String
```

Esto preserva con inteligencia la naturaleza puramente OO de Ruby (las funciones siguen siendo métodos de objetos, donde el receptor implícito es **self**), a la vez que proporciona funciones que se pueden escribir de igual forma que en lenguajes tradicionales.

Una disciplina mental común en la programación OO, que ya se señaló en un capítulo anterior, tiene que ver con la separación de la *especificación* y la *implementación* o *qué* tareas se supone que un objeto realiza y *cómo* realmente se consiguen. El trabajo interno de un objeto debe mantenerse, por lo general, oculto a sus usuarios; sólo se tiene que preocupar de lo que entra y lo que sale y confiar en que el objeto sabe lo que está realizando internamente. Así, es generalmente útil que las clases posean métodos que el mundo exterior no ve, pero que se utilizan internamente (y que pueden ser mejorados por el programador cuando desee, sin modificar la forma en que los usuarios ven los objetos de esa clase). En el trivial ejemplo que sigue, piénsese que **engine** es el motor interno de la clase.

```
ruby> class Test
ruby|   def times_two(a)
ruby|           print a," dos veces es ",engine(a),"\n"
ruby|   end
```

```
ruby|   def engine(b)
ruby|           b*2
ruby|   end
ruby|   private:engine # esto oculta engine a los usuarios
ruby| end
Test
ruby> test = Test.new
#<Test:0x401c4230>
ruby> test.engine(6)
ERR: (eval):1: private method ‘engine’ called for #<Test:0x401c4230>
ruby> test.times_two(6)
6 dos veces es 12
nil
```

Se podría esperar que **test.engine(6)** devolviese 12, pero por el contrario se nos comunica que **engine** es inaccesible cuando actuamos como usuario del objeto **Test**. Sólo otros métodos de **Test**, como **times_two** tienen permiso para utilizar **engine**. Se nos obliga a pasar por el interfaz público, que es el método **times_two**. El programador que está al cargo de la clase puede modificar **engine** (en este caso cambiando **b*2** por **b+b** suponiendo que así mejora el rendimiento) sin afectar cómo los usuarios interactúan con el objeto **Test**. Este ejemplo, por supuesto, es demasiado simple para ser útil; los beneficios de control de accesos se manifiestan cuando se comienzan a crear clases más complicadas e interesantes.

Capítulo 17. Métodos singleton

El comportamiento de una instancia viene determinado por su clase, pero hay veces que sabemos que una determinada *instancia* debe tener un comportamiento especial. En la mayoría de los lenguajes debemos meternos en la problemática de crear otra clase e instanciarla sólo una vez. En Ruby se puede asignar a cada OBJETO sus propios métodos.

```
ruby> class SingletonTest
ruby|   def size
ruby|           print "25\n"
ruby|   end
ruby| end
nil
ruby> test1 = SingletonTest.new
#<SingletonTest:0x401c4604>
ruby> test2 = SingletonTest.new
#<SingletonTest:0x401c4514>
ruby> def test2.size
ruby|   print "10\n"
ruby| end
nil
ruby> test1.size
25
nil
ruby> test2.size
10
nil
```

En este ejemplo, **test1** y **test2** pertenecen a la misma clase, pero a **test2** se le ha redefinido el método **size** y por lo tanto se comportan de forma diferente. Un método que pertenece sólo a un objeto se conoce como *método singleton*.

Los métodos singleton se utilizan frecuentemente en los elementos de un interfaz gráfico de usuario (GUI[1]) cuando se deben realizar acciones diferentes cuando se pulsan botones diferentes.

Los métodos singleton no son únicos de Ruby, aparecen también en CLOS, Dylan, etc. Otros lenguajes como por ejemplo Self y NewtonScript, sólo tienen métodos singleton. A estos se les conoce como lenguajes *basados en prototipos*

Notas

1. Graphical User Interface

Capítulo 18. Módulos

Los módulos en Ruby son similares a las clases, excepto en:

- Un módulo no puede tener instancias
- Un módulo no puede tener subclases
- Un módulo se define con **module ... end**

Ciertamente ... la clase Module de un módulo es la superclase de la clase Class de una clase. ¿Se pilla esto? ¿No? Sigamos.

Existen dos usos típicos de los módulos. Uno es agrupar métodos y constantes relacionadas en un repositorio central. El módulo **Math** de Ruby hace esta función:

```
ruby> Math.sqrt(2)
1.414213562
ruby> Math::PI
3.141592654
```

El operador **::** indica al intérprete de Ruby qué módulo debe consultar para obtener el valor de la constante (es concebible, que algún otro módulo a parte de **Math** interprete **PI** de otra forma). Si queremos referenciar a los métodos o constantes de un módulo, directamente, sin utilizar **::**, podemos *incluir* ese módulo con **include**:

```
ruby> include Math
Object
ruby> sqrt(2)
1.414213562
ruby> PI
3.141592654
```

El otro uso de los módulos se denomina *mixin*. Algunos lenguajes OO, incluidos el C++, permiten *herencia múltiple*, es decir, una clase puede heredar de más de una superclase. Un ejemplo de herencia múltiple en el mundo real es un despertador, se podría pensar que un despertador es una clase de *reloj* y que también pertenece a la clase de objetos que podríamos llamar *zumbadores*.

Ruby, con toda la intención del mundo, no implementa herencia múltiple real, aunque la técnica de los *mixins* es una buena alternativa. Recuérdese que los módulos no se pueden instanciar ni se pueden crear subclases de ellos; pero si se incluye un módulo en la definición de una clase sus métodos quedan añadidos a ella, es decir se asocian (*mixin*[1]) a la clase.

Se puede pensar que los mixins son una forma de pedir qué propiedades concretas se desean. Por ejemplo, si una clase tiene un método **each** funcional, asociarla con el módulo **Enumerable** de la biblioteca estándar nos proporciona gratuitamente los métodos **sort** y **find**.

Esta utilización de los módulos proporciona la funcionalidad básica de la herencia múltiple permitiéndonos representar las relaciones de la clase en una simple estructura en árbol que simplifica considerablemente la implementación del lenguaje (Los diseñadores de Java hicieron una elección parecida).

Notas

1. asociar

Capítulo 19. Objetos procedimiento

A menudo es deseable tener la posibilidad de definir respuestas específicas a sucesos inesperados. Resulta que esto se consigue con gran sencillez si podemos pasar un bloque de código a otros métodos, lo que significa que deseamos tratar el código como si fuesen datos.

Un *objeto procedimiento* nuevo se obtiene utilizando **proc**:

```
ruby> quux = proc {
ruby|   print "QUUXQUUXQUUX!!!\n"
ruby| }
#<Proc:0x401c4884>
```

Ahora **quux** referencia a un objeto y como las mayoría de los objetos, tiene un comportamiento que se puede invocar. Concretamente, podemos pedir que se ejecute a través de su método **call**

```
ruby> quux.call
QUUXQUUXQUUX!!!
nil
```

Luego, después de todo esto. ¿Podemos utilizar **quux** cómo un argumento de un método? Ciertamente.

```
ruby> def run ( p )
ruby|   print "Vamos a llamar a un procedimiento ... \n"
ruby|   p.call
ruby|   print "Finalizado. \n"
ruby| end
nil
ruby> run quux
Vamos a llamar a un procedimiento ...
QUUXQUUXQUUX!!!
Finalizado.
nil
```

El método **trap** nos permite asignar una respuesta personalizada a cualquier señal del sistema.

```
ruby> inthandler = proc{ print "^C ha sido pulsado.\n" }
#<Proc:0x401c4104>
ruby> trap "SIGINT", inthandler
nil
```

Normalmente, al pulsar ^C se sale del intérprete. Ahora se imprime un mensaje y el intérprete sigue ejecutándose, así no se pierde el trabajo realizado. (No nos encontramos atrapados en el intérprete para siempre; todavía se puede salir tecleando **exit** o pulsando ^D.)

Una observación final antes de pasar a otros temas: no es necesario dar al objeto procedimiento un nombre antes de asociarlo a una señal. Un objeto procedimiento *anónimo* equivalente se asemejaría a:

```
ruby> trap "SIGINT", proc{ print "^C ha sido pulsado.\n" }
#<Proc:0x401c4104>
```

O de una forma más compacta todavía,

```
ruby> trap "SIGINT", ’print "^C ha sido pulsado.\n"’
#<Proc:0x401c3d44>
```

Este formato abreviado es mas adecuado y legible cuando se escriben pequeños procedimientos anónimos.

Capítulo 20. Variables

Ruby tiene tres clases de variables, una clase de constante y exactamente dos pseudo-variables. Las variables y las constantes no tienen tipo. Aunque las variables sin tipo tienen sus inconvenientes, presentan más ventajas y se adaptan mejor a la filosofía *rápido y sencillo* de Ruby

En la mayoría de los lenguajes hay que declarar las variables para especificar su tipo, si se pueden modificar (e.g. si son constantes) e indicar su ámbito, ya que no es ningún problema el tipo y como vamos a ver, el resto se obtiene a partir del nombre, en Ruby no se necesita declarar las variables.

El primer carácter de un identificador lo cataloga de un plumazo:

Tabla 20-1. Clases de variables

$	Variable global
@	Variable instancia
[a-z] ó _	Variable local
[A-Z]	Constante

Las únicas excepciones a lo expuesto en la tabla son las pseudo-variables de Ruby: **self**, que referencia al objeto que está en ese momento en ejecución y **nil** que es el valor nulo que toman las variables no inicializadas. Ambos tienen un identificador como de variable local pero **self** es una variable global que la mantiene el interprete y **nil** es una constante. Como estas son las únicas excepciones no provocan mucha confusión.

No se debe asignar valores a **self** y **nil** principalmente porque un valor de **self** referencia al objeto de nivel superior:

```
ruby> self
main
ruby> nil
nil
```

20.1. Variables globales

Una variable global tiene un nombre que comienza con **$**. Se puede utilizar en cualquier parte de un programa. Antes de inicializarse, una variable global tiene el valor especial **nil**.

```
ruby> $foo
nil
ruby> $foo = 5
5
ruby> $foo
5
```

La variables globales deben utilizarse con parquedad. Son peligrosas porque se pueden modificar desde cualquier lugar. Una sobreutilización de variables globales puede dificultar la localización de errores; también indica que no

se ha pensado detenidamente el diseño del programa. Siempre que se encuentre la necesidad de utilizar una variable global, hay que darle un nombre descriptivo para que no se pueda utilizar inadvertidamente para otra cosa (Llamarle **$foo** como se ha hecho en el ejemplo es probablemente una mala idea)

Una característica notable de las variables globales es que se pueden trazar; se puede definir un procedimiento que se llame cada vez que se modifique el valor de la variable.

```
ruby> trace_var:$x, proc{print "$x es ahora ", $x, "\n"}
nil
ruby> $x = 5
$x es ahora 5
5
```

Cuando una variable global se la atavía para que funcione con un disparador que se llama cada vez que se modifica, se la conoce como *variable activa*. Son útiles, por ejemplo, para mantener un GUI actualizado.

Existe un grupo especial de variables cuyos nombres constan del símbolo del dolar (**$**) seguido de un carácter. Por ejemplo, **$$** contiene el número de identificación del proceso del intérprete de Ruby, y es de sólo lectura. A continuación se muestran las principales variables del sistema y su significado (acudir al **manual de referencia** de Ruby para más detalles)

Tabla 20-2. Variables de sistema

$!	Último mensaje de error
$@	Posición del error
$_	Última cadena leída con **gets**
$.	Último **número** de línea leído por el interprete
$&	Última cadena que ha coincidido con una expresión regular
$~	Última cadena que ha coincidido con una expresión regular como array de subexpresiones
$*n*	La *n-ésima* subexpresión regular de la última coincidencia (igual que **$~[n]**)
$=	flag para tratar igual las mayúsculas y minúsculas
$/	Separador de registros de entrada
$\	Separador de registros de salida
$0	El nombre del fichero del guión Ruby
$*	El comando de la línea de argumentos
$$	El número de identificación del proceso del intérprete Ruby
$?	Estado de retorno del último proceso hijo ejecutado

De las variables anteriores **$_** y **$~**, tienen ámbito local. Sus nombres sugieren que deberían tener ámbito global, pero son más útiles de esta forma y existen razones históricas para utilizar estos identificadores.

20.2. Variables de instancia

Una variable de instancia tiene un nombre que comienza con **@** y su ámbito está limitado al objeto al que referencia **self**. Dos objetos diferentes, aún cuando pertenezcan a la misma clase, pueden tener valores diferentes en sus variables de instancia. Desde el exterior del objeto, las variables de instancia, no se pueden alterar e incluso, no se pueden observar (es decir, en Ruby las variables de instancia nunca son públicas) a excepción de los métodos proporcionados explícitamente por el programador. Como con las variables globales, las variables de instancia tienen el valor **nil** antes de que se inicialicen

Las variables de instancia en Ruby no necesitan declararse. Esto da lugar a una estructura flexible de los objetos. De hecho, cada variable de instancia se añade dinámicamente al objeto la primera vez que se la referencia

```
ruby> class InstTest
ruby|   def set_foo(n)
ruby|           @foo = n
ruby|   end
ruby|   def set_bar(n)
ruby|           @bar = n
ruby|   end
ruby| end
nil
ruby> i = InstTest.new
#<InstTest:0x401c3e0c>
ruby> i.set_foo(2)
2
ruby> i
#<InstTest:0x401c3e0c  @foo=2>
ruby> i.set_bar(4)
4
ruby> i
#<InstTest:0x401c3e0c  @bar=4, @foo=2>
```

Obsérvese que **i** no informa del valor de **@bar** hasta que no se haya llamado al método **set_bar**

20.3. Variables locales

Una variable local tiene un nombre que empieza con una letra minúscula o con el carácter de subrayado (_). Las variables locales no tienen, a diferencia de las variables globales y las variables de instancia, el valor **nil** antes de la inicialización:

```
ruby> $foo
nil
ruby> @foo
nil
ruby> foo
ERR: (eval):1: undefined local variable or method ‘foo’ for #<Object:0x401d2c90>
```

La primera asignación que se realiza sobre una variable local actúa como una declaración. Si se referencia a una variable local no inicializada, el intérprete de Ruby piensa que se trata de una llamada a un método con ese nombre; de ahí el mensaje de error del ejemplo anterior.

Generalmente el ámbito de una variable local es uno de los siguientes:

- **proc{ ... }**
- **loop{ ... }**
- **def ... end**
- **class ... end**
- **module ... end**
- Todo el programa (si no es aplicable ninguno de los puntos anteriores)

En el siguiente ejemplo **define?** es un operador que verifica si un identificador está definido. Si lo está, devuelve una descripción del mismo, en caso contrario, devuelve **nil**. Como se ve el ámbito de **bar** es local al bucle, cuando se sale del bucle, **bar** está sin definir.

```
ruby> foo =44; print foo, "\n"; defined? foo
44
"local-variable"
ruby> loop{bar = 45;print bar, "\n"; break}; defined? var
45
nil
```

Los objetos procedimiento que residen en el mismo ámbito comparten las variables locales que pertenecen a ese ámbito. En el siguiente ejemplo, la variable **bar** es compartida por **main** y los objetos procedimiento **p1** y **p2**:

```
ruby> bar=0
0
ruby> p1 = proc{|n| bar = n}
#<Proc:0x401c3e34>
ruby> p2 = proc{bar}
#<Proc:0x401c3cf4>
ruby> p1.call(5)
5
ruby> bar
5
ruby> p2.call
5
```

Obsérvese que no se puede omitir la línea **bar=0** inicial; esta asignación es la que garantiza que el ámbito de **bar** incluirá a **p1** y **p2**. Si no, **p1** y **p2** tendrán al final cada uno su propia variable local **bar** y la llamada a **p2** dará lugar a un error de “variable o método no definido”.

Una característica muy poderosa de los objetos procedimiento se deriva de su capacidad para recibir argumentos; las variables locales compartidas permanecen válidas incluso cuando se las pasa fuera de su ámbito original.

```
ruby> def box
ruby|   contents = 15
ruby|   get = proc{contents}
ruby|   set = proc{|n| contents = n}
```

```
ruby|   return get, set
ruby| end
nil
ruby> reader, writer = box
[#<Proc:0x401c33d0>, #<Proc:0x401c33bc>]
ruby> reader.call
15
ruby> writer.call(2)
2
ruby> reader.call
2
```

Ruby es especialmente inteligente con respecto al ámbito. En el ejemplo, es evidente que la variable **contents** está compartida por **reader** y **writer**. Ahora bien, es posible definir varios pares reader-writer que utilicen **box** cada uno de los cuales comparten su propia variable **contents** sin interferir uno con otro.

```
ruby> reader_1, writer_1 = box
[#<Proc:0x401c2e6c>, #<Proc:0x401c2e58>]
ruby> reader_2, writer_2 = box
[#<Proc:0x401c2cdc>, #<Proc:0x401c2cc8>]
ruby> writer_1.call(99)
99
ruby> reader_1.call
99
ruby> reader_2.call
15
```

Capítulo 21. Constantes

Una constante tiene un nombre que comienza con una letra mayúscula. Se le debe asignar valor sólo una vez. En la implementación actual de Ruby, reasignar un valor a una constante genera un aviso y no un error (la versión no ANSI de eval.rb no informa de este aviso):

```
ruby> fluid = 30
30
ruby> fluid = 31
31
ruby> Solid = 32
32
ruby> Solid = 33
(eval):1: warning: already initialized constant Solid
33
```

Las constantes se pueden definir en una clase, pero a diferencia de las variables de instancia, son accesibles desde el exterior de la misma.

```
ruby> class ConstClass
ruby|   C1=101
ruby|   C2=102
ruby|   C3=103
ruby|   def show
ruby|           print C1," ",C2, " ",C3,"\n"
ruby|   end
ruby| end
nil
ruby> C1
ERR: (eval):1: uninitialized constant C1
ruby> ConstClass::C1
101
ruby> ConstClass.new.show
101 102 103
nil
```

Las constantes también se pueden definir en un módulo.

```
ruby> module ConstModule
ruby|   C1=101
ruby|   C2=102
ruby|   C3=103
ruby|   def showConstants
ruby|           print C1," ",C2," ",C3,"\n"
ruby|   end
ruby| end
nil
ruby> C1
ERR: (eval):1: uninitialized constant C1
```

```
ruby> include ConstModule
Object
ruby> C1
101
ruby> showConstants
101 102 103
nil
ruby> C1=99 # realmente una idea no muy buena
99
ruby> C1
99
ruby> ConstModule::C1 # La constante del módulo queda sin tocar ...
101
ruby> ConstModule::C1=99
ERR: (eval):1: compile error
(eval):1: parse error
ConstModule::C1=99
              ^
ruby> ConstModule::C1 #... independientemente de lo que hayamos jugado con ella
101
```

Capítulo 22. Procesamiento de excepciones: rescue

Un programa en ejecución puede encontrarse con problemas inesperados. Podría no existir un fichero que desea leer, al salvar algunos datos se podría llenar un disco, un usuario podría introducir algún tipo de datos de entrada poco adecuados.

```
ruby> file = open("algun_fichero")
ERR: (eval):1:in ‘open’: No such file or directory - "algun_fichero"
```

Un programa robusto manejará estas situaciones prudente y elegantemente. Satisfacer estas expectativas puede ser una tarea exasperante. Los programadores en C se supone que deben verificar toda llamada al sistema que pudiese fallar y decidir inmediatamente que hacer.

```
FILE *file = fopen("algun_fichero","r");
if (file == NULL) {
  fprintf(stderr, "No existe el fichero\n");
  exit(1);
}
bytes_read = fread(buf,1,bytes_desired,file);
if (bytes_read != bytes_desired) {
  /* aquí, más gestión de errores ... */
}
...
```

Con esta práctica tan aburrida los programadores tienden a ser descuidados y la incumplen siendo el resultado un programa que no gestiona adecuadamente las excepciones. Por otro lado, si se realiza adecuadamente, los programas se vuelven ilegibles debido a que hay mucha gestión de errores que embrolla el código significativo.

En Ruby, como en muchos lenguajes modernos, se pueden gestionar las excepciones para bloques de código de una forma compartimentalizada, lo que permite tratar los imprevistos de forma efectiva sin cargar excesivamente ni al programador ni a cualquier otra persona que intente leer el código posteriormente. El bloque de código marcado con **begin** se ejecutará hasta que haya una excepción, lo que provoca que el control se transfiera a un bloque con el código de gestión de errores, aquel marcado con **rescue**. Si no hay excepciones, el código de **rescue** no se usa. El siguiente método devuelve la primera línea de un fichero de texto o **nil** si hay una excepción.

```
def first_line( filename )
  begin
    file = open(filename)
    info = file.gets
    file.close
    info # Lo último que se evalúa es el valor devuelto
  rescue
    nil # No puedo leer el fichero, luego no devuelvo una cadena
  end
end
```

A veces nos gustaría evitar con creatividad un problema. A continuación, si el fichero no existe, se prueba a utilizar la entrada estándar:

```
begin
  file = open("algun_fichero")
rescue
  file = STDIN
end
begin
  # ... procesamos la entrada ...
rescue
  # ... aquí tratamos cualquier otra excepción
end
```

Dentro del código de **rescue** se puede utilizar **retry** para intentar de nuevo el código en **begin**. Esto nos permite reescribir el ejemplo anterior de una forma más compacta:

```
fname = "algun_fichero"
begin
  file = open(fname)
  # ... procesamos la entrada ...
rescue
  fname = "STDIN"
  retry
end
```

Sin embargo, este ejemplo tiene un punto débil. Si el fichero no existe este reintento entrará en un bucle infinito. Es necesario estar atento a estos escollos cuando se usa **retry** en el procesamiento de excepciones.

Toda biblioteca Ruby genera una excepción si ocurre un error y se pueden lanzar excepciones explícitamente dentro del código. Para lanzar una excepción utilizamos **raise**. Tiene un argumento, la cadena que describe la excepción. El argumento es opcional pero no se debería omitir. Se puede acceder a él posteriormente a través de la variable global especial **$!**.

```
ruby> begin
ruby|   raise "error"
ruby| rescue
ruby|   print "Ha ocurrido un error: ", $!, "\n"
ruby| end
Ha ocurrido un error: error
nil
```

Capítulo 23. Procesamiento de excepciones: ensure

Cuando un método termina su ejecución puede que se necesiten ciertas labores de limpieza. Quizás se tenga que cerrar un fichero abierto, se deban liberar los datos de los buffers, etc. Si siempre hubiese un único punto de salida para cada método, se podría poner con seguridad todo el código de limpieza en un lugar concreto y saber que se ejecutará, sin embargo, un método podría retornar en varios lugares o el código de limpieza podría saltarse inesperadamente debido a una excepción.

```
begin
  file = open("/tmp/algun_fichero","w")
  # ... Escribimos en el fichero ...
  file.close
end
```

En el ejemplo superior, si ocurre una excepción durante la parte del código en que se escribe en el fichero, éste quedará abierto. Si no deseamos recurrir al siguiente tipo de redundancia:

```
begin
  file = open("/tmp/algun_fichero","w")
  # ... Escribimos en el fichero ...
  file.close
rescue
  file.close
  fail # levantamos una excepción
end
```

Que es desgarbada y se nos escapa de las manos cuando el código se vuelve más complicado debido a que hay que tratar todo **return** y **break**.

Por esta razón se añadió otra palabra reservada a esquema **begin ... rescue ... end**, **ensure**. El bloque de código de **ensure** se ejecuta independientemente del éxito o fracaso del bloque de código en **begin**.

```
begin
  file = open("/tmp/algun_fichero","w")
  # ... Escribimos en el fichero ...
rescue
  # ... gestionamos las excepciones ...
ensure
  file.close # ... Y esto se ejecuta siempre.
end
```

Se puede utilizar **ensure** sin **rescue** y viceversa, pero si se utilizan en el mismo bloque **begin ... end**, **rescue** debe preceder a **ensure**.

Capítulo 24. Accesores

En un capítulo anterior se trató brevemente las variables instancia, pero no se hizo mucho con ellas. Las variables instancia de un objeto son sus atributos, eso que diferencia a un objeto de otro dentro de la misma clase. Es importante poder modificar y leer estos atributos; lo que supone definir métodos denominados *accesores de atributos*. Veremos en un momento que no siempre hay que definir los métodos accesores explícitamente, pero vayamos paso a paso. Los dos tipos de accesores son los de *escritura* y los de *lectura*.

```
ruby> class Fruta
ruby|   def set_kind(k) # escritor
ruby|     @kind = k
ruby|   end
ruby|   def get_kind    # lector
ruby|     @kind
ruby|   end
ruby| end
nilx
ruby> f1 = Fruta.new
#<Fruta:0x401c4410>
ruby> f1.set_kind("melocotón")      #utilizamos el escritor
"melocotón"
ruby> f1.get_kind               #utilizamos el lector
"melocotón"
ruby> f1                        #inspeccionamos el objeto
#<Fruta:0x401c4410  @kind="melocotón">
```

Sencillo; podemos almacenar y recuperar información sobre la clase de fruta que queremos tener en cuenta. Pero los nombres de nuestros métodos son un poco largos. Los siguientes son más breves y convencionales:

```
ruby> class Fruta
ruby|   def kind=(k)
ruby|     @kind = k
ruby|   end
ruby|   def kind
ruby|     @kind
ruby|   end
ruby| end
nil
ruby> f2 = Fruta.new
#<Fruta:0x401c30c4>
ruby> f2.kind = "banana"
"banana"
ruby> f2.kind
"banana"
```

24.1. El método inspect

En estos momentos es adecuada una pequeña disgresión. Ya se habrá notado que cuando deseamos ver directamente un objeto se nos muestra algo críptico como lo siguiente **#<Fruta:0x401c30c4>**. Este es un comportamiento por defecto que se puede modificar. Todo los que se necesita es definir un método denominado **inspect**. Éste puede devolver una cadena que describa el objeto de una forma razonable, incluyendo el estado de alguna o todas las variables instancia.

```
ruby> class Fruta
ruby|   def inspect
ruby|     "una fruta de la variedad " + @kind
ruby|   end
ruby| end
nil
ruby> f2
una fruta de la variedad banana
```

Un método relacionado es **to_s** (convertir a cadena) que se utiliza al imprimir un objeto. En general se puede pensar que **inspect** es una herramienta para cuando se escriben y depuran programas, y **to_s** una forma de refinar la salida de un programa. **eval.rb** utiliza **inspect** cuando muestra resultados. Se puede utilizar el método **p** para obtener con sencillez resultados para la depuración de programas.

```
# las dos líneas siguientes son equivalentes
p anObject
print anObject.inspect, "\n"
```

24.2. Facilitando la creación de accesores

Dado que muchas variables instancia necesitan métodos accesores, Ruby proporciona abreviaturas para las formas convencionales.

Tabla 24-1. Accesores

Abreviatura	Efecto
attr_reader :v	**def v; @v; end**
attr_writer :v	**def v=(value); @v=value; end**
attr_accesor :v	**attr_reader :v; attr_writer :v**
attr_accesor :v, :w	**attr_accesor :v; attr_accessor :w**

Tomemos ventaja de esto y añadamos información fresca. Primero pediremos la generación de un escritor y un lector y luego incorporaremos la nueva información en **inspect**.

```
ruby> class Fruta
ruby|   attr_accessor :condition
ruby|   def inspect
```

```
ruby|     "una " + @kind + " " + @condition
ruby|   end
ruby| end
nil
ruby> f2.condition = "madura"
"madura"
ruby> f2
una banana madura
```

24.3. Más diversión con la fruta

Si nadie se come nuestra fruta madura, quizás es momento de que pague su precio.

```
ruby> class Fruta
ruby|   def time_passes
ruby|           @condition = "podrida"
ruby|   end
ruby| end
nil
ruby> f2
una banana madura
ruby> f2.time_passes
"podrida"
ruby> f2
una banana podrida
```

Pero mientras estábamos jugando con esto se ha introducido un pequeño problema. ¿Qué ocurre si intentamos crear una tercera pieza de fruta en estos momentos? Recuérdese que las variables instancia no existen hasta que no se les asigne valor.

```
ruby> f3 = Fruta.new
ERR: failed to convert nil into String
```

El que se queja es el método **inspect** y con motivos. Se le ha indicado que informe sobre el tipo y la condición de una pieza de fruta, pero **f3** no tiene asignado ninguno de sus atributos. Si se quiere, es posible redefinir este método para que compruebe que las variables instancia están definidas (utilizando el método **defined?**) e informar de ellos sólo si es así aunque esto puede que no sea de mucha utilidad dado que toda pieza de fruta es de un tipo y está en una determinada condición, parece que se debiera asegurar que los atributos se definen de alguna forma. Este es el tema del siguiente capítulo.

Capítulo 25. Inicialización de objetos

La clase Fruta del capítulo anterior tiene dos variables instancia, una para describir la clase de fruta y otra para describir su estado. Después de redefinir el método **inspect** de la clase, nos dimos cuenta de que no tiene sentido que una pieza de fruta carezca de esas características. Afortunadamente, Ruby tiene un mecanismo para asegurar que las variables instancia se inicialicen siempre.

25.1. El método initialize

Siempre que Ruby crea un objeto nuevo, busca un método llamado **initialize** y lo ejecuta. Luego lo más sencillo que se puede hacer es utilizar este método para dar valores a las variables instancia, así el método **inspect** no tiene nada por lo que quejarse.

```
ruby> class Fruta
ruby|   def initialize
ruby|     @kind = "manzana"
ruby|     @condition = "madura"
ruby|   end
ruby| end
nil
ruby> f4 = Fruta.new
una manzana madura
```

25.2. Modificando suposiciones por requisitos

Hay veces en las que no tiene mucho sentido la presencia de valores por defecto. ¿Existe una cosa tal como una fruta por defecto? Es preferible que se deba especificar el tipo en el momento de la creación de cada pieza de fruta. Para hacer esto se debe añadir un argumento formal al método **initialize**. Por razones en las que no vamos a entrar, los argumentos que se entregan a **new** se pasan a **initialize**

```
ruby> class Fruta
ruby|   def initialize(k)
ruby|     @kind = k
ruby|     @condition = "madura"
ruby|   end
ruby| end
nil
ruby> f5 = Fruta.new "pera"
una pera madura
ruby> f6 = Fruta.new
ERR: (eval):1:in ‘initialize’: wrong # of arguments(0 for 1)
```

25.3. Inicialización flexible

Hemos visto que una vez que se asocia un argumento al método **initialize** no se puede omitir sin que se genere un error. Si queremos ser más considerados podemos utilizar el argumento si se proporciona, y en caso contrario, recurrir al valor por defecto.

```
ruby> class Fruta
ruby|   def initialize(k="manzana")
ruby|     @kind = k
ruby|     @condition = "madura"
ruby|   end
ruby| end
nil
ruby> f5 = Fruta.new "pera"
una pera madura
ruby> f6 = Fruta.new
una manzana madura
```

Se pueden utilizar los valores por defecto de un argumento en cualquier método no sólo en **initialize**. Los argumentos hay que organizarlos de tal forma que aquellos con valores por defecto aparezcan al final de la lista.

A veces es útil tener varias formas de inicializar un objeto. Aunque está fuera del ámbito de este tutorial, Ruby permite reflexión sobre los objetos y listas de argumentos de tamaño variable. Ambas técnicas combinadas permiten realizar sobrecarga de métodos.

Capítulo 26. Entresijos

Este capítulo trata algunos problemas prácticos.

26.1. Delimitadores de sentencias

Algunos lenguajes requieren algún tipo de puntuación, a menudo el punto y coma (;), para finalizar toda sentencia de un programa. Por el contrario, Ruby recurre al convenio seguido por shells como **sh** y **csh**. Varias sentencias en una línea se han de separar con puntos y comas sin que se necesite al final de la línea; LF se trata como un punto y coma. Si una línea termina en \ (backslash) se ignora el LF que le sigue; lo que permite tener una única línea lógica que comprende varias líneas físicas.

26.2. Comentarios

¿Por qué escribir comentarios? Aunque el buen código tiende a ser auto-descriptivo, a menudo es útil realizar comentarios en el margen. Es un error creer que otras personas que examinen el código comprendan inmediatamente lo que se pretendía hacer. Aparte, y desde una perspectiva práctica, quién de nosotros no ha tenido que realizar una corrección o mejora en un programa después de un cierto periodo de tiempo y decir: he escrito esto, pero ¿qué demonios se supone que hace?

Algunos programadores experimentados señalarán, con bastante razón, que comentarios contradictorios o desactualizados pueden ser peor que ningún comentario en absoluto. Evidentemente, los comentarios no deben ser un sustituto de un código legible; si el código es poco claro, es probable que también sea erróneo. Es probable que se necesite comentar más cuando se está aprendiendo Ruby y menos cuando se llegue a expresar las ideas en código sencillo, elegante y legible.

Ruby sigue el convenio, común entre los lenguajes de guiones, de utilizar el símbolo de la almohadilla para indicar el comienzo de un comentario. El interprete ignora cualquier cosa que siga a una almohadilla, que no esté entre comillas, hasta el final de la línea en la que aparece

Para facilitar la escritura de grandes bloques de comentarios el interprete también ignora cualquier cosa comprendida entre una línea inicial con **=begin** y una final con **=end**.

```
#!/usr/bin/ruby

=begin
  ************************************************************************
  Este es un bloque de comentarios, algo que se escribe en beneficio de
  los lectores (incluido uno mismo). El interprete lo ignora. No hay
  necesidad de utilizar '#' al comienzo de cada línea
  ************************************************************************
=end
```

26.3. Organización del código

El intérprete de Ruby procesa el código conforme lo lee. No existe nada semejante a una fase de compilación; si algo no se ha leído todavía, sencillamente está sin definir.

```
# Este código da lugar al error "undefined method":

print successor(3),"\n"

def successor(x)
  x + 1
wend
```

Como cabría esperar a primera vista esto no fuerza a que se deba organizar el código de un modo estrictamente bottom-up. Cuando el intérprete encuentra la definición de un método puede incluir con seguridad referencias no definidas, siempre y cuando se asegure que se definirán antes de llamar realmente al método:

```
# Conversión de fahrenheit a celsius, dividida en dos pasos

def f_to_c(f)
  scale (f - 32.0) # Referencia adelantada, pero es correcto
end

def scale(x)
  x * 5.0 / 9.0
end

printf "%.1f es una temperatura agradable.\n",f_to_c( 72.3 )
```

Aunque pueda parecer un poco menos adecuado que lo que se suele usar en Perl o Java, es menos restrictivo que intentar escribir código C sin prototipos (lo que obliga a mantener siempre una ordenación parcial de quién referencia a quién). Poner el código de más alto nivel al final del fichero, funciona siempre. Y esto no es una gran molestia aunque a primera vista lo pudiese parecer. Una forma sensata e indolora de conseguir el comportamiento que se desea es definir una función **main** al principio del fichero y llamarla al final.

```
#!/usr/bin/ruby

def main
  # Aquí el código de nivel superior
end

  # ... Todo el código de apoyo aquí, organizado como se crea más adecuado ...

main # ... y se inicia la ejecución aquí.
```

También sirve de ayuda que Ruby proporcione herramientas para dividir programas complicados en bloques legibles, reutilizables, y relacionados lógicamente. Se ha visto la utilización de **include** para acceder a módulos. Pero también pueden ser útiles **load** y **require**. **load** funciona como si el fichero al que referencia fuese copiado y pegado (algo parecido a la directiva **#include** del preprocesador C). **require** es un poco más sofisticada, carga el código como

mucho sólo una vez y cuando se necesite. Existen otras diferencias entre **load** y **require**; para más información se puede acudir el manual del lenguaje o a la FAQ.

26.4. Esto es todo

Este tutorial debería se suficiente para arrancar y escribir programas en Ruby. Si surgen más preguntas se puede bucear en el *manual de referencia* para aprender Ruby con más detalle. También son fuentes importantes de recursos la *FAQ* y la *biblioteca de referencia*

¡Suerte y felices codificaciones!