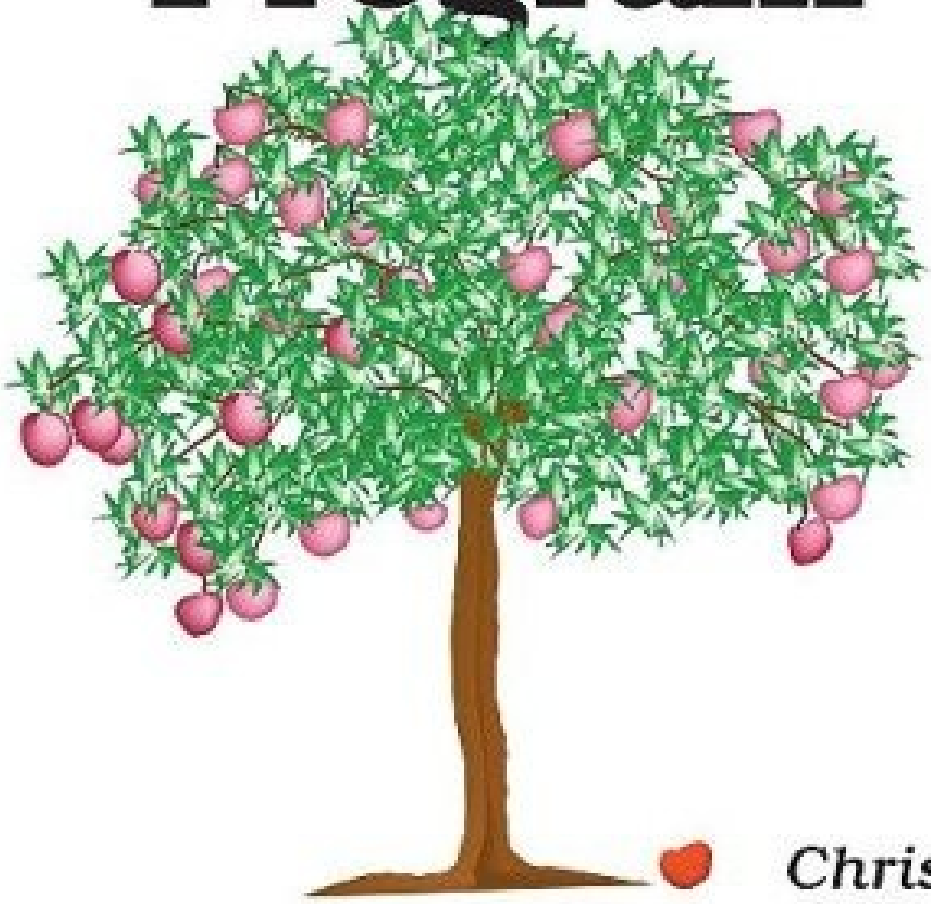
The Pragmatic Programmers

Learn to Program



Chris Pine

The Facets of Ruby Series

Inicio

Cuando programas una computadora, necesitas "hablar" en la forma que tu computadora entienda: un lenguaje de programación. Existen diferentes lenguajes y muchos de ellos son excelentes. En esta guía voy a usar mi lenguaje de programación favorito: *Ruby*

Además de ser mi favorito, Ruby es también el lenguaje más fácil de aprender que he visto (y eso que he visto unos cuantos) De hecho, esa es mi verdadera motivación para escribir esta guía: No decidí escribir una guía y luego escogí Ruby porque es mi lenguaje favorito; fue la simplicidad de Ruby lo que me motivó a escribirla. (Esribir una guía similar en otro lenguaje, como C++ o Java, hubiese requerido cientos y cientos de páginas) ¡Pero no creas que Ruby es un lenguaje para principiantes solo porque es fáci! Es un lenguaje poderoso y de calibre profesional, si es que existe alguno así.

Cuando escribes algo en un lenguaje humano, lo que se escribe es texto. Cuando escribes algo en un lenguaje de computadora, lo que se escribe es

código

. He incluido muchos ejemplos de código Ruby en esta guía, muchos de ellos son programas completos que puedes ejecutar en tu equipo. Para hacer el código más fácil de leer, he marcado partes del código de distintos colores y formatos.

Si encuentras algo que no entiendas o si tienes una pregunta que no has podido resolver, ¡escríbela y sigue leyendo! Es posible que la respuesta llegue en un capítulo posterior. Sin embargo, si tu pregunta no ha sido resuelta al finalizar la guía, te voy a mostrar donde preguntar. Hay muchas personas dispuestas a ayudar; solo necesitas saber donde están.

Pero primero necesitas descargar e instalar Ruby en tu computadora.

Instalación en Windows

La instalación de Ruby en Windows es rápida, primero debes descargar el Instalador de Ruby. Puede que haya unas cuantas versiones para escoger; en este tutorial se utilizará la versión 1.9.3-p194, así que asegúrate de por lo menos descargar una version tan reciente como la indicada. Luego simplemente ejecuta el programa de instalación. A menos que tengas una buena razón, sería bueno que lo instales en la ubicación por defecto.

Para poder programar, vas a necesitar escribir tus programas y ejecutarlos. Para esto vas a necesitar un editor de texto y la línea de comando.

El instalador de Ruby viene con un editor de texto llamado SciTE (el editor de texto Scintilla). Puedes ejecutar SciTE seleccionandolo del menú de arranque. Si quieres que tu código esté coloreado como los ejemplos de esta guía, descarga estos archivos y ponlos en tu carpeta de SciTE (`c:/ruby/scite` si escogiste la ubicación por defecto):

- Propiedades Globales
- Propiedades de Ruby

Sería una buena idea crear una carpeta para guardar ahí todos tus programas. Asegúrate de guardar tus programas en esta carpeta.

Para acceder a tu línea de comando, selecciona

Menú de Inicio - Accesorios - Símbolo del Sistema

Luego navega hacia la carpeta designada para tus programas. Escribir #{input 'cd ..'} te llevará una carpeta arriba (carpeta padre), y #{input 'cd foldername'} te llevará a la carpeta llamada `foldername`. Para ver todos los folders en tu carpeta actual, escribe #{input 'dir /ad'}.

¡Eso es todo! Estás listo para aprender a programar.

Instalación en Mac

Si tienes Mac OS X 10.7 (Lion), ¡Ruby está instalado en tu sistema! ¿Qué puede ser más fácil? Desafortunadamente si tienes una versión anterior a Max OS X 10.1 no creo que puedas usar Ruby.

Para poder programar, vas a necesitar escribir tus programas y ejecutarlos. Para esto vas a necesitar un editor de texto y la línea de comando.

Tu línea de comando es accesible a través del programa Terminal (que encontrarás en Applications/Utilities).

Como editor de texto puedes usar uno que te sea familiar o donde te sientas cómodo. Si usas TextEdit, ¡asegúrate de guardar los programas como texto! Si no, *no funcionarán*. Otras opciones para programar son emacs, vi o pico, todos accesibles desde la línea de comando.

¡Eso es todo! Estás listo para aprender a programar.

Linux Installation

Primero, debes revisar si ya tienes Ruby instalado. Escribe `which ruby`. Si dice algo como `/usr/bin/which: no ruby in (...)`, entonces necesitas descargar Ruby, si no, revisa que versión de Ruby tienes instalado escribiendo `ruby -v`. Si es más antigua que la última versión estable mostrada en la página de descarga mencionada arriba, deberías actualizarla.

Si eres el usuario root, seguro no necesitas ningunas instrucciones para instalar Ruby. Si no lo eres, debes pedirle a tu administrador de redes que lo instale por ti. (De esa forma todos los usuarios del sistema podrán usar Ruby.)

Por otro lado, puedes simplemente instalarlo para que tu lo puedas usar. Mueve el archivo que descargaste a una carpeta temporal, como `$HOME/tmp`. Si el nombre del archivo es `ruby-1.9.3-p194.tar.bz2`, puedes abrirlo ejecutando `tar zxvf ruby-1.9.3-p194.tar.bz2`. Cambia de directorio al que acabas de crear (en este ejemplo, `cd ruby-1.9.3-p194`).

Configura la instalación escribiendo `./configure --prefix=$HOME`. Luego escribe `make`, lo que compilará el interpretador de Ruby. Esto tomará unos minutos. Después de terminar, escribe `make install` para instalarlo.

Luego, vas a necesitar agergar `$HOME/bin` a tu ruta de búsqueda de comandos editando el archivo `$HOME/.bashrc`. (Tal vez debas cerrar y abrir una nueva sesión para que esto tenga efecto.) Al terminar, prueba tu instalación: #{input 'ruby -v'}. Si te dice la versión que tienes instalada, puedes eliminar los archivos en `$HOME/tmp` (o donde sea que los hayas puesto).

¡Eso es todo! Estás listo para aprender a programar.

Números

Ahora que tienes todo instalado ¡vamos a escribir un programa! Abre tu editor de texto favorito y escribe lo siguiente:

```
puts 1+2
```

Guarda tu programa (sí, ¡eso es un programa!) como `calc.rb` (el **.rb** es lo que usualmente ponemos al final de los programas escritos con Ruby). Ahora ejecuta tu programa escribiendo `ruby calc.rb` en la línea de comandos. Deberías obtener un `3` en tu pantalla. ¿Lo ves?, programar no es tan difícil, ¿cierto?

Introducción a `puts`

¿Cómo es que funciona ese programa? Seguramente puede adivinar qué es lo que `1+2` hace; nuestro programa es básicamente lo mismo que:

```
puts 3
```

`puts` simplemente escribe en la pantalla lo que sea que escribamos a continuación.

Enteros y flotantes

En la mayoría de los lenguajes de programación (y Ruby no es la excepción) los números sin decimales son llamados *enteros* (traducción de *integers* en su versión en Inglés), y los números con punto decimal normalmente son llamados *números de punto flotante* (o llamados solo *flotantes* de ahora en adelante ya que proviene de su original en inglés *floats*).

Aquí hay algunos enteros:

```
5
-205
999999999999999999999999
0
```

Y aquí hay algunos flotantes:

```
54.321
```

```
0.001
-205.3884
0.0
```

En la práctica, la mayoría de los programas no usan flotantes, sólo enteros. (Después de todo, nadie quiere leer 7.4 emails, o navegar 1.8 páginas, o escuchar 5.24 de sus canciones favoritas...) Los flotantes se usan más con propósitos académicos (como por ejemplo experimentos científicos) y para gráficos en 3D. Incluso la mayoría de los programas financieros usan enteros, ¡simplemente llevan un registro de los centavos!

Aritmética simple

Hasta ahora, tenemos todo lo que necesita una calculadora simple. (Las calculadoras siempre usan flotantes, así que si quiere que su computadora actúe como una calculadora, también debería usar flotantes). Para suma y resta, usamos `+` y `-`, como ya lo vimos. Para la multiplicación, usamos `*`, y para la división usamos `/`. La mayoría de los teclados tienen estas teclas en el teclado numérico a la derecha. Intentemos expandir un poco nuestro programa calc.rb. Escribe lo siguiente y ejecútalo:

```
puts 1.0 + 2.0
puts 2.0 * 3.0
puts 5.0 - 8.0
puts 9.0 / 2.0
```

Esto es lo que retorna el programa:

```
3.0
6.0
-3.0
4.5
```

(Los espacios en el programa no son importantes; simplemente hacen que el código sea más fácil de leer). Bueno, eso no fue muy sorprendente. Ahora probemos con enteros:

```
puts 1+2
puts 2*3
puts 5-8
puts 9/2
```

Básicamente lo misno, ¿no?

```
3
6
-3
4
```

Uh... ¡excepto por el último! Pero cuando se hace aritmética con enteros, se obtienen enteros. Cuando su computadora no puede obtener la respuesta "correcta", siempre redondea hacia abajo. (Por supuesto, `4` *es* la respuesta correcta en aritmética con enteros para `9/2`; simplemente tal vez no sea el resultado que esperaba).

Tal vez se pregunte para qué es útil la división entera. Bueno, digamos que va al cine, pero sólo tiene $9. Aquí en Portland, puede ver una película en el Bagdad por $2. ¿Cuántas películas puede ver allí? `9/2`... `4` películas. 4.5 definitivamente *no* es la respuesta correcta ya que no le dejarán ver la mitad de una película o dejar que una mitad suya vea la película entera... algunas cosas simplemente no son divisibles.

¡Así que ahora experimente con algunos programas por su cuenta! Si quiere escribir expresiones más complejas, puede usar paréntesis. Por ejemplo:

Código:

```
puts 5 * (12-8) + -15
puts 98 + (59872 / (13*8)) * -52
```

Resultado:

```
5
-29802
```

Algunas cosas por intentar

Escribe un programa que te diga:

- cuántas horas hay en un año?
- cuántos minutos hay en una década?
- ¿cuántos segundos de edad tiene usted?
- ¿cuántos chocolates espera comer en su vida?
- **Advertencia:** ¡Esta parte del programa puede tomar tiempo para computarse!

Aquí hay una pregunta más difícil:

- Si tengo 1031 millones de segundos de edad, ¿cuántos años tengo?

Cuando termine de jugar con números, hechemos un vistazo a algunos textos.

Textos

Así que hemos aprendido todo acerca de números, ¿pero qué acerca de letras? ¿palabras? ¿textos?

Nos referimos a grupos de letras en un programa como *textos*. (Tú puedes pensar en letras impresas siendo ensartadas en un cartel.) Aquí hay algunos textos:

```
'Hola.'
'Ruby la rompe.'
'5 es mi número favorito... ¿Cuál es el tuyo?'
'Snoopy dice #%^?&*@! cuando le aplastan el dedo del pie.'
'     '
''
```

Como puedes ver, los textos pueden tener puntuación, dígitos, símbolos, y espacios dentro... más que solo palabras. Ese último texto no tiene nada y podemos llamarlo un texto *vacío*.

Hemos estado usando `puts` para imprimir números, intentémoslo con textos:

Código:

```
puts 'Hola, mundo!'
puts ''
puts 'Adiós.'
```

Resultado:

```
Hola Mundo

Adiós
```

Eso funcionó bien. Ahora inténtalo con algún texto tuyo.

Aritmética de textos

Igual que hacer hacer aritmética con números, ¡tú puedes también hacer aritmética con textos! Bueno, algo así... puedes sumar textos, digamos. Intentemos sumar dos textos y ver que hace `puts` con eso.

Código:

```
puts 'Me gusta' + 'el pastel de manzana.'
```

Resultado:

```
Me gustael pastel de manzana
```

Whoops! Me olvidé de poner un espacio entre `'Me gusta'` y `'el pastel de manzana.'`. Los espacios no importan generalmente salvo si lo hacen dentro de los textos. (Es verdad lo que dicen: las computadoras no hacen lo que tú *quieres* que hagan, solo lo que tú *le dices* que hagan) Intentémoslo nuevamente:

Código:

```
puts 'Me gusta ' + 'el pastel de manzana.'
puts 'Me gusta' + ' el pastel de manzana.'
```

Resultado:

```
Me gusta el pastel de manzana
Me gusta el pastel de manzana
```

(Como puedes ver, no importó a que texto le agregué el espacio)

Así que puedes sumar textos, ¡pero también puedes multiplicarlas! (Por un número ...) Observa esto:

Código:

```
puts 'parpadeo ' * 4
```

Resultado:

```
batea tus párpados
```

(Solo bromeaba, en realidad muestra esto):

```
parpadeo parpadeo parpadeo parpadeo
```

Si lo piensas, tiene todo el sentido. Después de todo, `7*3` realemnte solo significa `7+7+7`, así que `'moo'*3` solo significa `'moo'+'moo'+'moo'`.

`12` vs `'12'`

Antes de ir más allá, debemos asegurarnos de entender la diferencia entre *números* y *dígitos*. `12` es un número, pero `'12'` es un texto de dos dígitos.

Juguemos con esto un poco:

Código:

```
puts  12  +  12
puts '12' + '12'
puts '12  +  12'
```

Resultado:

```
24
1212
12  +  12
```

Y qué sucede con esto?:

Código:

```
puts  2  *  5
puts '2' *  5
puts '2  *  5'
```

Resultado:

```
10
22222
2  *  5
```

Estos ejemplos fueron bastante directos. De cualquier modo, si no tienes cuidado en cómo mezclas tus textos y tus números podrías encontrarte con...

Problemas

En este punto podrías haber probado algunas cosas que *no funcionaron*. Si no lo has hecho, aquí hay algunas:

Código:

```
puts '12' + 12
```

```
puts '2' * '5'
```

Resultado:

```
can't convert Fixnum into String (TypeError)
```

Hmmm... un mensaje de error. El problema es eque en realidad no puedes sumar un número a un texto, o multiplicar un texto por otro texto. No tiene más sentido que esto:

```
puts 'Betty' + 12
puts 'Fred' * 'John'
```

Algo más para tener cuidado: Puedes escribir `'pig'*5` en un programa, dado que sólo significa `5` veces el texto `'pig'` todo sumado. De cualquier modo, tú *no puedes* escribir `5*'pig'`, ya que eso significa `'pig'` veces el número `5`, lo cual es simplemente tonto.

Finalmente, y si quisieras que tu programa imprimiera `¡Mi nombre es O'hara!`? Podrías intentar:

```
puts '¡Mi nombre es O'hara!'
```

Bueno, *eso* no funcionará; No intentaré ni ejecutarlo. La computadora pensó que habíamos terminado con el texto. (Esta es la razón de porque es lindo tener un editor de texto que haga *coloreo sintáctico* por ti) ¿Entonces, cómo permitimos que la computadora sepa que queremos permanecer en el texto? Tenemos que *escapar* el apóstrofe, de esta manera:

```
puts '¡Mi nombre es O\'hara!'
```

La barra invertida es el carácter de escape. En otras palabras, si tú tienes una barra invertida y otro carácter, ellos a veces son convertidos a otro carácter. Las únicas cosas que una barra invertida escapa son los apóstrofes y las propias barras invertidas. (Si lo piensas un poco, los caracteres escapados siempre se escapan a sí mismo) Unos pocos ejemplos vendrían muy bien, me parece:

Código:

```
puts '¡Mi nombre es O\'hara!'
puts 'barra invertida al final del texto:  \\'
puts 'arriba\\abajo'
puts 'arriba\abajo'
```

Resultado:

```
¡Mi nombre es O'hara!
barra invertida al final del texto:  \
arriba\abajo
arriba\abajo
```

Dado que la barra invertida *no* escapa un `'d', pero *si* si se escapa a si misma, esos dos últimos caracteres son idénticos. No parecen lo mismo en el código, pero en tu computadora son realmente los mismos.

Si tienes alguna pregunta, solo ¡sigue leyendo! No podría contestar cada pregunta en *esta* página, después de todo.

Variables

Hasta ahora, cuando usamos `'puts'` para un texto o número, esto desaparece. A lo que me refiero es que, si queremos imprimir algo dos veces, necesitamos escribirlo dos veces:

Código:

```
puts '...puedes decir eso de nuevo...'
puts '...puedes decir eso de nuevo...'
```

Resultado:

```
...puedes decir eso de nuevo...
...puedes decir eso de nuevo...
```

Sería bueno si pudiesemos escribirlo solo una vez y quedárnoslo... guardarlo en algún lado. Bueno, si podemos, por supuesto; de otra manera, ¡no lo hubiese mencionado!

Para guardar el texto en la memoria de tu computadora, necesitamos darle un nombre al texto. Los programadores se refieren frecuentemente a

Para guardar el texto en la memoria de tu computadora, necesitamos darle un nombre al texto. Los programadores se refieren frecuentemente a este proceso como **asignación** y llaman a los nombres **variables**. Esta variable puede ser cualquier secuencia de letras o números, pero el primer caracter necesita ser minúscula. Probemos nuestro programa de nuevo, pero esta vez voy a darle el nombre `'myString'` al texto (aunque podría haber usado cualquier nombre, por ejemplo `'str'` or `'myOwnLittleString'` or `'henryTheEighth'`).

Código:

```
myString = '...puedes decir eso de nuevo...'
puts myString
puts myString
```

Resultado:

```
...puedes decir eso de nuevo...
...puedes decir eso de nuevo...
```

En todo momento cuando haces referencia a `'myString'`, el programa usa en su reemplazo `"'...puedes decir eso de nuevo...'"`. Puedes pensar en la variable `'myString'` como "apuntando" al texto `"'...puedes decir eso de nuevo...'"`. Este es un ejemplo un poco más interesante.

Código:

```
name = 'Patricia Rosanna Jessica Mildred Oppenheimer'
puts 'Me llamo ' + name + '.'
puts 'Wow!  "' + name + '" es un nombre realmente largo!'
```

Resultado:

```
Me llamo Patricia Rosanna Jessica Mildred Oppenheimer
Wow1 Patricia Rosanna Jessica Mildred Oppenheimer es un nombre realmente largo!
```

También, así como podemos **asignar** un objeto a una variable, podemos **reasignar** un objeto diferente a esa variable. (Por eso es que las llamamos variables: porque varían.)

Código:

```
composer = 'Mozart'
puts composer + ' fue "el amo", en su día.'

composer = 'Beethoven'
puts 'Pero yo prefiero a ' + composer + ', personalmente.'
```

Resultado:

```
Mozart fue "el amo", en su día.
Pero yo prefiero a Beethoven, personalmente.
```

Por supuesto, las variables pueden apuntar a cualquier tipo de objecto, no solo texto:

Código:

```
var = 'solo otro ' + 'texto'
puts var

var = 5 * (1+2)
puts var
```

Resultado:

```
solo otro texto
15
```

De hecho, las variables pueden apuntar a casi cualquier cosa... excepto otras variables. ¿Pero que pasa si lo intentamos?

Código:

```
var1 = 8
var2 = var1
puts var1
puts var2
```

```
puts ''

var1 = 'ocho'
puts var1
puts var2
```

Resultado:

```
8
8

ocho
8
```

Cuando primero tratamos de apuntar `var2` a `var1` en realidad apuntamos a `8` (que es el valor al cual apuntaba `var1`) Luego cuando apuntamos `var1` a `'ocho'`, el valor de `var2`no cambia ya que en realidad no estaba apuntando a `var1` sino a `8`. Ahora que tenemos variables, números y textos, vamos a ver como mezclarlas.

Mezclando

Hemos revisado distintos tipos de objetos (números y textos) y hemos hecho que variables apunten a ellos; lo que queremos hacer después es que funcionen todos juntos.

Hemos visto que si queremos que un programa imprima `25`, lo que sigue ***no*** funciona, porque no puedes sumar números y texto:

Código:

```
var1 = 2
var2 = '5'

puts var1 + var2
```

Resultado:

```
in `+': String can't be coerced into Fixnum (TypeError)
```

Parte del problema es que tu computadora no sabe si estabas tratando de obtener `7` (`2 + 5`) o si querías obtener `'25'` (`'2' + '5'`).

Antes de poderlos sumar, necesitamos alguna forma de conseguir la representación textual (en 'letras') de `var1`, o la representación numérica de `var2`.

Conversiones

Para obtener la representación en texto de un objeto, simplemente escribimos `.to_s` después de el objeto:

Código:

```
var1 = 2
var2 = '5'

puts var1.to_s + var2
```

Resultado:

```
25
```

De la misma manera, `to_i` devuelve la versión numérica entera de un objeto y `to_f` devuelve la versión de punto flotante, o sea la que lleva decimales. Veamos que hacen estos tres métodos (y lo que *no* hacen) un poco más de cerca:

Código:

```
var1 = 2
var2 = '5'

puts var1.to_s + var2
puts var1 + var2.to_i
```

Resultado:

```
25
7
```

Nota que, inclusive después de que obtuvimos la versión en texto de `var1` al llamar `to_s`, `var1` siempre apuntó a `2`, y nunca a `'2'`. A menos que reasignes explícitamente `var1` (lo que requiere de un signo `=`), apuntará a `2` durante la duración del programa.

Ahora probemos conversiones más interesantes (y un poco raras):

Código:

```
puts '15'.to_f
puts '99.999'.to_f
puts '99.999'.to_i
puts '5 es mi numero favorito.'.to_i
puts '¿Quién pregunta acerca de 5 o lo que sea?'.to_i
puts 'Tu mamá hizo.'.to_f
puts 'fibroso'.to_s
puts 3.to_i
```

Resultado:

```
15.0
99.999
99
5
0
0.0
fibroso
3
```

Esto probablemente te ha sorprendido un poco. El primero es bastante estándar, devolviendo `15.0`. Después de eso, convertimos el texto `'99.999'` a un número de punto flotante y a un entero. El primero hizo lo que esperábamos; el entero fue, como siempre, redondeado hacia abajo.

Luego, tuvimos ejemplos de algunos textos... *inusuales* ... convertidos a números `to_i` ignora lo primero que no entienda, y el resto del texto a partir de ahí. Así que lo primero va a ser convertido a `5`, pero los otros, ya que comenzaban con letras, fueron ignorados completamente... así que la computadora simplemente elige usar cero.

Finalmente, vimos que las últimas dos conversiones no hacían nada, tal como esperábamos.

Hay algo extraño en nuestro método favorito... Mira esto:

Código:

```
puts 20
puts 20.to_s
puts '20'
```

Resultado:

```
20
20
20
```

¿Por qué estas tres líneas retornan el mismo resultado? Bueno, las últimas dos deberían, ya que `20.to_s` *es* `'20'`. Pero, ¿qué pasa con la primera, el entero `20`? Para tal caso, ¿qué significa escribir *el entero* 20? Cuando escribes *2* y luego *0* en un papel, estas escribiendo texto, no un entero. *El entero* 20 es el número de dedos que tengo; no es *2* seguido de un *0*.

Bueno, ahí está el secreto de nuestro amigo, `puts`: Antes de que `puts` trate de imprimir un objecto, usa `to_s` para obtener la versión en texto de ese objeto. De hecho, la *s* en `puts` significa *string* (que en español significa texto); `puts` realmente significa *put string* (que en español significa imprimir texto).

Esto puede no parecer muy emocionante ahora, pero hay muchos, *muchos* objetos en Ruby (¡inclusive vas a aprender a crear tus propios objetos!), y es bueno saber que va a pasar cuando trates de ejecutar `puts` con un objeto bastante raro, como la foto de tu abuela, o un archivo de música, etc. Pero eso vendrá después...

Mientras tanto, tenemos unos cuantos métodos para ti, y ellos nos permiten escribir todo tipo de programas divertidos...

Los Métodos `gets` y `chomp`

Si `puts` significa *put string*, estoy seguro que puedes adivinar que significa `gets` (recibir texto). Y así como `puts` siempre devuelve texto, `gets` solo funciona con texto. ¿Y de donde lo recibe?

¡De ti! Bueno, de tu teclado, en realidad. Ya que tu teclado solo escribe texto, todo funciona bien. Lo que realmente pasa es que `gets` simplemente espera, leyendo todo lo que tipeas hasta que presionas `Enter`. Intentémoslo:

Código:

```
puts gets
```

Resultado:

```
*¿Hay eco aquí?*
¿Hay eco aquí?
```

Por supuesto, lo que sea que escribas simplemente será impreso en pantalla por ti. Ejecútalo unas cuantas veces e intenta escribir cosas distintas.

¡Ahora podemos escribir programas interactivos! En este caso, escribe tu nombre y te saludará:

```
puts 'Hola, ¿cuál es tu nombre?'
name = gets
puts '¿Tu nombre es ' + name + '?  ¡Es un nombre adorable!'
puts 'Encantado de conocerte, ' + name + '.  :)'
```

Uhm! Acabo de ejecutarlo, escribí mi nombre y esto es lo que he obtenido:

```
Hola, ¿cuál es tu nombre?
*Chris*
¿Tu nombre es Chris
?  ¡Es un nombre adorable!
Encantado de conocerte, Chris
.  :)
```

Hmmm... parece que cuanto escribí las letras *C*, *h*, *r*, *i*, *s*, y luego presioné `Enter`, `gets` recibió todas las letras de mi nombre *y* el `Enter`! Afortunadamente, hay un método para este tipo de cosas: `chomp`. Esto elimina cualquier `Enter` al final del texto. Intentémoslo de nuevo, pero con `chomp` para que nos ayude:

Código:

```
puts 'Hola, ¿cuál es tu nombre?'
name = gets.chomp
puts '¿Tu nombre es ' + name + '?  ¡Es un nombre adorable!'
puts 'Encantado de conocerte, ' + name + '.  :)'
```

Resultado:

```
Hola, ¿cuál es tu nombre?
Chris
¿Tu nombre es Chris?  ¡Es un nombre adorable!
Encantado de conocerte, Chris.  :)
```

¡Mucho mejor! Nota que ya que `name` apunta a `gets.chomp`, no tenemos que usar `name.chomp`; `name` ya fue procesado por `chomp.

Algunas cosas por intentar

- Escribe un programa que pregunte por el nombre de una persona, luego el segundo nombre y luego el apellido. Finalemente, debería saludar a la persona con el nombre completo.
- Escribe un programa que pregunte por el número favorito del usuario. Has que tu programa agregue un número, luego sugiera el resultado como el número favorito pero *más grande y mejorado*. (Pero hazlo con tacto.)

Una vez que hayas terminado los dos programas (y otros que hayas intentado), tratemos de aprender más (y más sobre) ¡métodos!.

Más acerca de Métodos

Mas acerca de Metodos

Hemos visto diferentes métodos: `puts`, `gets`, etc. (***Prueba:*** *Listar todos los métodos que hemos visto hasta ahora!. Hay diez de ellos, la respuesta está abajo*), pero no hemos realmente hablado sobre que hacen los métodos. Sabemos que hacen, pero no lo que son.

Pero realmente, esto *es* lo que son: cosas que generan otras. Si objetos (como textos, enteros y flotantes) son los sujetos en el lenguaje Ruby, entonces los métodos son como verbos. Y, justo como en español, tú no puedes tener un verbo sin un sustantivo para *hacer* algo. Por ejemplo, tic-tac no es algo que solo ocurre; un reloj (o algo) tiene que hacer esto. En español podemos decir: "El reloj hace tic-tac". En Ruby podemos decir `clock.tick` (asumiendo por supuesto que `clock` es un objeto Ruby). Los programadores pueden decir que estamos "llamando el método `tick` de `clock`" o llamamos al "`tick` de `clock`".

Entonces, ¿has hecho la prueba? Bien. Bueno, estoy seguro que recordaras los métodos `puts`, `gets`, y `chomp`, dado que ya hablamos sobre ellos. Probablemente también recuerdas los métodos de conversión `to_i`, `to_f`, y `to_s`. Sin embargo, ¿has visto los otros cuatro? Porque, estos no son otros que nuestros viejos amigos para la aritmética ¡`+`, `-`, `*`, y `/`!

Entonces, como cada verbo necesita un sustantivo, entonces cada método necesita un objeto. Esto es generalmente fácil de indicar: es el que viene justo antes de un punto, como nuestro ejemplo `clock.tick`, o en `101.to_s`. Algunas veces, sin embargo, esto no es tan obvio; como con los métodos aritméticos. Como resulta, `5 + 5` es solo otra forma fácil de escribir `5.+ 5`. Por ejemplo:

Código:

```
puts 'hola '.+ 'mundo'
puts (10.* 9).+ 9
```

Resultado:

```
hola mundo
99
```

Esto no es muy lindo, por lo que no vamos a escribir siempre como ahora; sin embargo, es importante para entender que sucede *realmente*.

Esto también nos da un profundo entendimiento de porque podemos hacer `'pig'*5` pero no podemos hacer `5*'pig'`: `'pig'*5` esta diciendo a `'pig'` de hacer de multiplicando, pero `5*'pig'` esta diciendo a `5` de hacer de multiplicando. `'pig'` sabe como hacer `5` copias de si mismo y agregar todos ellos juntos; sin embargo, `5` tendrá mucha mas dificultad en tiempo de hacer `'pig'` copias de si *mismo* y sumarlos a todos juntos.

Y, por supuesto, continuaremos teniendo `puts` y `gets` para explicar. ¿Dónde están sus objetos? En español, puedes algunas veces dejar fuera el sustantivo; por ejemplo, si un villano grita "¡Muere!", el sustantivo implícito es a quien él esta gritando. En Ruby, si digo `puts 'ser o no ser'`, lo que realmente estoy diciendo es `self.puts 'ser o no ser'`. Entonces ¿que es `self`? Esta es una variable especial que apunta al objeto en
el que estás. No siempre sabemos como estar *en* un objeto, pero hasta que nos demos cuenta, siempre iremos a estar en un gran objeto que es... ¡el programa entero! (sin embargo en este caso no es posible llamarlo en forma explícita) Observa lo siguiente:

Código:

```
NoPuedoCreerQueUnaVariableConNombreTanLargoApunteA3 = 3
puts NoPuedoCreerQueUnaVariableConNombreTanLargoApunteA3
self.puts NoPuedoCreerQueUnaVariableConNombreTanLargoApunteA3
```

Resultado:

```
3
in `<main>': private method `puts' called for main:Object (NoMethodError)
```

Si no alcanzaste a comprender todo, está bien. Lo importante es todo método está siendo propiedad de un objeto, incluso si no tiene un punto enfrente de este. Si entiendes esto estás preparado.

Métodos imaginativos acerca de Strings

Vamos a aprender unos pocos pero interesantes métodos. No tienes porque memorizar todos; puedes mirar esta pagina de nuevo si te olvidas de alguno. Yo solo quiero mostrarte una *pequeña* parte de lo que puede hacer un texto. De hecho, no recuerdo ni siquiera la mitad de los métodos para textos; pero está bien, porque hay buenas referencias en internet con todo acerca de textos listados y explicados. (Voy a mostrarte donde encontrar esas referencias al final del tutorial.) Realmente, tampoco *quiero* saber todo acerca de los métodos para texto; sino sería como tratar de conocer cada palabra en el diccionario. Puedo hablar español bien sin conocer cada una de las palabras del diccionario... ¿y no es ese realmente el objetivo del diccionario? Entonces ¿no *tienes* que conocer que hay en éste?

Entonces, nuestro primer método para texto es `reverse`, el cual nos una version invertida de un texto:

Código:

```
var1 = 'parar'
var2 = 'subrayado'
var3 = 'Puedes pronunciar esta oración al reves?'

puts var1.reverse
puts var2.reverse
puts var3.reverse
puts var1
puts var2
puts var3
```

Resultado:

```
rarap
odayarbus
?sever la nóicaro atse raicnunorp sedeuP
parar
subrayado
Puedes pronunciar esta oración al reves?
```

Como puedes ver, `reverse` no revierte el orden en el string original; este solo hace una nueva versión de éste en reversa. Esto es porque `var1` continua `'parar'` aún después de que llamamos `reverse` sobre `var1`.

Otro método para texto es `length`, el cual nos dice el numero de caracteres (incluyendo caracteres) en el texto:

Código:

```
puts 'Cuál es tu nombre completo?'
name = gets.chomp
puts '¿Sabes que hay ' + name.length + ' caracteres en tu nombre, ' + name + '?'
```

Resultado:

```
Cuál es tu nombre completo?
Christopher David Pine
#<TypeError: can't convert Fixnum into String>
```

¡¡Uhh!! Algo salió mal, y esto parece que ocurrió después la línea `name = gets.chomp`... ¿Puedes ver el problema? Fijate si puedes darte cuenta.

El problema es con `length`: esto te devuelve un número, pero nosotros queremos un texto. Esto es fácil, necesitamos solo agregar `to_s` (y cruzar nuestros dedos):

Código:

```
puts 'Cuál es tu nombre completo?'
name = gets.chomp
puts '¿Sabías que hay ' + name.length.to_s + ' caracteres en tu nombre, ' + name + '?'
```

Resultado:

```
Cuál es tu nombre completo?
Christopher David Pine
Sabías que hay 22 caracteres en tu nombre, Christopher David Pine
```

No, no conocía esto. **Nota:** esto es el número de *caracteres* en mi nombre, no el número de *letras*. Supongo que podríamos escribir un programa el cual nos pregunte por nuestro primer nombre, segundo nombre y apellidos individualmente, y entonces sumar estos tamaños todos juntos... hey, ¡porque no haces esto! Comienza, esperaré.

¿Lo hiciste? ¡Bien! Es un lindo programa, ¿no? Después de unos pocos capítulos más, pienso, estarás sorprendido de lo que podrás hacer.

Entonces, hay también un número de métodos para texto los cuales cambian el contenido (mayúsculas y minúsculas) de tu texto. `upcase` cambian cada minúscula por mayúscula. `swapcase` cambia en cada letra en el string("Hola".swapcase #=> "hOLA"), y finalmente, `capitalize` es como `downcase`, excepto que esto cambia solo el primer carácter a mayúsculas(si es una letra).

Código:

Código:

```
letters = 'aAbBcCdDeE'
puts letters.upcase
puts letters.downcase
puts letters.swapcase
puts letters.capitalize
puts ' a'.capitalize
puts letters
```

Resultado:

```
AABBCCDDEE
aabbccddee
AaBbCcDdEe
Aabbccddee
 a
aAbBcCdDeE
```

Esto es bastante estándar. Como puedes ver desde la linea `puts ' a'.capitalize`, el método `capitalize` solo deja en mayúsculas el primer *carácter*, no la primera *letra*. También, como hemos visto antes, en todas estas llamadas a métodos, `letters` permanece igual. No quiero decir que se dedica solo a esto, pero es importante entenderlo. Hay algunos métodos los cuales *hacen* cambios a los objetos asociados, pero no los hemos visto aún, y no lo haremos por algún tiempo.

El último tipo de métodos que veremos son los de formato visual. El primero es, `center`, suma espacios al comienzo y final para hacer que este centrado. Sin embargo, solo tienes que decir `puts` a lo que quieres imprimir, y `+` a lo que quieres sumar, pero tienes que decir a `center` que ancho tiene que tener el string centrado. Entonces si quiero centrar las lineas de un poema, debería hacer algo como esto:

Código:

```
lineWidth = 50
puts(                'Old Mother Hubbard'.center(lineWidth))
puts(               'Sat in her cupboard'.center(lineWidth))
puts(         'Eating her curds an whey,'.center(lineWidth))
puts(          'When along came a spider'.center(lineWidth))
puts(         'Which sat down beside her'.center(lineWidth))
puts('And scared her poor shoe dog away.'.center(lineWidth))
```

Resultado:

```
          Old Mother Hubbard
         Sat in her cupboard
      Eating her curds an whey,
       When along came a spider
      Which sat down beside her
  And scared her poor shoe dog away.
```

Mmmm.. no pienso que esto es un campamento de verano, pero estoy muy cansado para buscar esto. (Entonces, quise alinear la parte `.center lineWidth` , entonces puse esos espacios extras antes de los textos. Esto es así solo porque pienso que es más lindo de esta forma. Los programadores generalmente tienen duros conceptos acerca de que es lindo en un programa, y a menudo confrontan acerca de esto. Cuanto más programes, más lograras tu propio estilo.) Hablando de ser perezoso a la hora de programar, lo cual no es siempre algo malo en programación. Por ejemplo, fijate como guardé el ancho del poema en la variable `lineWidth`? Esto es que si entonces quiero regresar más tarde y hacer el poema más ancho, solo tengo que cambiar la variable al comienzo del programa, antes que en cada línea. Con un poema muy largo, esto podría ahorrarme un montón de tiempo. Este tipo de pereza es realmente una virtud en programación.

Entonces, acerca del centrado... tú te darás cuenta que esto no es muy lindo como podría serlo un procesador de texto. Si realmente quieres un perfecto centrado (y quizás una fuente mas linda), entonces deberías ¡solo usar un procesador de textos!. Ruby es una herramienta maravillosa, pero no la herramienta correcta para *cualquier* trabajo.

Los otros dos métodos de formato de textos son `ljust` y `rjust`, lo cual significan *justificado izquierdo* y *justificado derecho*. Estos son similares a `center`, excepto que ellos rellenan los lados derecho e izquierdo respectivamente. Vamos a verlos en acción:

Código:

```
lineWidth = 40
str = '--> text <--'
puts str.ljust  lineWidth
puts str.center lineWidth
puts str.rjust  lineWidth
puts str.ljust(lineWidth/2) + str.rjust(lineWidth/2)
```

Resultado:

```
--> text <--
              --> text <--
                            --> text <--
--> text <--                --> text <--
```

Algunas cosas por intentar

- Escribe un programa `Jefe Enojado`. Este debe preguntar de mala manera que quieres. Cualquier cosa que consultes, el Jefe Enojado deberá devolverte la consulta de mala forma, y luego despedirte. Por ejemplo, si tu escribes `Quiero un aumento.`, deberá contestarte `PERO QUE DICES HOMBRE "¿¡¿QUIERES UN AUMENTO."?!? ¡¡ESTAS DESPEDIDO!!`

- Entonces aquí hay algo para que puedas jugar un poco más con `center`, `ljust`, y `rjust`: Escribe un programa el cual muestre una Tabla de Contenidos que se parezca a lo siguiente:

Listado:

```
                Tabla de Contenidos

Capítulo 1:  Números                      página 1
Capítulo 2:  Letras                      página 72
Capítulo 3:  Variables                  página 118
```

Matemáticas Avanzadas

(Esta sección es totalmente opcional. Asume un conocimiento previo de matemáticas. Si no estás interesado, puedes ir directamente al siguiente capítulo Control de Flujo sin problemas. Aunque, una rápida vista de esta sección sobre ***Números aleatorios*** *debería venir bien.)*

No hay tantos métodos numéricos como los hay para textos (pienso que aun no los conozco a todos sin recurrir a la ayuda de documentación). Aquí, vamos a mirar el resto de los métodos aritméticos, un generador de números aleatorios, y el objeto `Math` , con sus métodos trigonométricos y transcendentales.

Más de aritmética

Los otros dos métodos aritméticos son `**` (potencia) y `%` (módulo). Entonces si quieres decir "cinco al cuadrado" en Ruby, deberías escribir algo así `5**2`. También puedes usar flotantes para tus exponentes, entonces si quieres una raíz cuadrada de 5, deberías escribir `5**0.5`. Los métodos módulo te dan el sobrante después de una división por un número. Entonces, por ejemplo, si divido 7 por 3, obtengo 2 con un remanente de 1. Vamos a ver como es que trabaja en un programa:

Código:

```
puts 5**2
puts 5**0.5
puts 7/3
puts 7%3
puts 365%7
```

Resultado:

```
25
2.23606797749979
2
1
1
```

De la última línea, aprendimos que un año (no bisiesto) tienen algún número de semanas, más un día. Entonces si tu cumpleaños fue un Martes este año, el próximo año será un Miércoles. Tu también puedes usar flotantes con los métodos módulo. Básicamente, funciona de una manera lógica... pero voy a mostrar un poco mas como trabajar con esto.

Hay un último método para mencionar antes de chequear el método random: `abs`. Este solo toma el valor absoluto de un número:

Código:

```
puts((5-2).abs)
puts((2-5).abs)
```

Resultado:

```
3
3
```

Números aleatorios

Ruby viene con un lindo generador de números aleatorios. El método para obtener un número aleatorio es `rand`. Si llamas `rand` , obtendrás un número flotante mayor o igual a `0.0` y menor a `1.0`. Si le proporcionas a `rand` un número entero (`5` por ejemplo), esto te devolverá un entero mayor o igual a `0` y menor a `5` (entonces son cinco números posibles, de `0` a `4`).

Vamos a ver `rand` en acción.

Código:

```
puts rand
puts rand
puts rand
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(1))
puts(rand(1))
puts(rand(1))
puts(rand(99999999999999999999999999999999999999999999999999999999999))
puts('El pronosticador del tiempo dijo que hay '+rand(101).to_s+'% chances de que llueva,')
puts('pero nunca debes confiar en el.')
```

Resultado:

```
0.866769322351658
0.155609260113273
0.208355946789083
61
46
92
0
0
0
229824775081318602319541087738875238616006939895184956998 62
El pronosticador del tiempo dijo que hay 47% chances de que llueva,
pero nunca debes confiar en el.
```

Fijate que utilicé `rand(101)` para obtener números entre `0` y `100`, y que el `rand(1)` siempre devuelve `0`. No entender el rango posible de retorno de valores es el error más grande que veo en gente que hace `rand`; aún programadores profesionales, más aún en productos finalizados que puedes comprar. Incluso tenía un reproductor de CD que si se configuraba en "Reproducción aleatoria," reproducia todas las canciones menos la última ... (Me pregunto ¿qué hubiera pasado si hubiera puesto un CD con sólo una canción en ella?)

Algunas veces querrás que `rand` retorne el *mismo* random de números incluso en la misma secuencia en dos diferentes ejecuciones de tu programa. (Por ejemplo, una vez estaba utilizando números generados aleatoriamente para crear un mundo al azar por un juego de computadoras. Encontré un mundo que realmente me gustó, quizás me hubiera gustado jugar de nuevo con éste o enviarlo a un amigo.) Con el fin de hacer esto, tu necesitas configurar la *"generación de éste"*, lo que se puede hacer con `srand`. Como lo siguiente:

Código:

```
srand 1776
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts ''
srand 1776
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
```

Resultado:

```
24
35
36
58
70

24
35
36
58
70
```

Esto hará la misma cosa cada vez que inicies con el mismo número. Si quieres obtener diferentes números(como pasaría si nunca utilizaras `srand`), entonces solo llamas a `srand 0`. Esto inicializa con un número realmente raro, utilizando (además de otras cosas) la hora actual de tu computadora, hasta los milisegundos.

El Objeto `Math`

Finalmente, vamos a echar un vistazo al objeto `Math`. Deberíamos ir directamente a este:

Código:

```
puts(Math::PI)
puts(Math::E)
puts(Math.cos(Math::PI/3))
puts(Math.tan(Math::PI/4))
puts(Math.log(Math::E**2))
puts((1 + Math.sqrt(5))/2)
```

Resultado:

```
3.14159265358979
2.71828182845905
0.5
1.0
2.0
1.61803398874989
```

La primera cosa de la que te habrás dado cuenta es probablemente de la notación `::`. Explicando el *alcance del operador* (que es esto) esta fuera del objetivo de este tutorial.

Como puedes ver, `Math` tiene todas las cosas que podrías esperar de una calculadora científica decente. Y como siempre, los flotantes están *realmente cerca* de ser la respuesta correcta.

Entonces ahora vamos a ver control de flujo!

Control de Flujo

Ahhhh, control de flujo. Acá es donde todo empieza a encajar. A pesar que este capítulo es más corto y facil que el capítulo de métodos, te abrirá un mundo nuevo de posibilidades para programar.

Luego de este capítulo, seremos capaces de escribir programas realmente interactivos; previamente hemos creado programas que *dicen* distintas cosas dependiente de lo que ingreses en tu teclado, pero luego de este capítulo estos también *harán* distintas cosas. Pero antes que podamos hacer eso, necesitamos ser capaces de comparar objectos en nuestros programas. Necesitamos...

Métodos de comparación

Avancemos rapidamente por esta sección para que podamos ir a la siguiente, **Ramificaciones**, donde todas las cosas geniales ocurren. Entonces, para ver si un objeto es mayor o menor que otro, usaremos los métodos > y <, así:

Código:

```
puts 1 > 2
puts 1 < 2
```

Resultado:

Resultado:

```
false
true
```

Sin problemas. De igual manera, podemos identificar si un objeto es mayor-o-igual-que otro o menor-o-igual-que con los métodos >= y <=

Código:

```
puts 5 >= 5
puts 5 <= 4
```

Resultado:

```
true
false
```

Finalmente, podemos ver si dos objetos son iguales o no utilizando == (que significa "¿son éstos iguales?") y != (que significa "¿son éstos diferentes?")

Es importante no confundir = con ==. = se utiliza para decirle a una variable a que objeto apuntar (asignación) y == se utiliza para responder la pregunta: "¿Son estos dos objetos iguales?"

Código:

```
puts 1 == 1
puts 2 != 1
```

Resultado:

```
true
true
```

Por supuesto, podemos comparar textos también. Cuando los textos se comparan, comparan su *ordenamiento lexicográfico*, que básicamente significa su orden en un diccionario. `gato` aparece antes que `perro` en el diccionario, así que:

Código:

```
puts 'gato' < 'perro'
```

Resultado:

```
true
```

Sin embargo, hay algo que tener en cuenta: la manera en que las computadoras normalmente asumen que las letras mayúsculas vienen antes que las letras minúsculas (Así es como almacenan los caracteres en las fuentes, por ejemplo: los caracteres en mayúscula primero y en minúscula después)

Esto significa que pensará que `'Zoológico'` aparece antes que `'ave'`, así que si quieres encontrar que palabra aparecería antes en un diccionario real, asegúrate de utilizar `downcase` (o `upcase` o `capitalize`) en ambas palabras antes de tratar de compararlas.

Un ultimo comentario antes de revisar **Ramificaciones**: Los métodos de comparación no nos están dando los textos `'true'` o `'false'` ("verdadero" y "falso" en español); nos están dando los objetos especiales `true` y `false`. (Por supuesto, , `true.to_s` nos devuelve `'true'`, que es por lo que `puts` imprimió `'true'`.)

`true` y `false` se utilizan todo el tiempo en...

Ramificaciones

Las Ramificaciones son un concepto simple, pero poderoso. De hecho, es tan simple que no necesito explicarlo, solo te lo mostraré:

Código:

```
puts 'Hola, ¿cual es tu nombre?'
name = gets.chomp
puts 'Hola, ' + name + '.'
if name == 'Chris'
  puts '¡Que nombre tan hermoso!'
end
```

```
end
```

si respondemos 'Chris':

```
Hola, ¿cual es tu nombre?
Chris
Hola, Chris.
¡Que nombre tan hermoso!
```

pero si ponemos otro nombre:

```
Hola, ¿cual es tu nombre?
Chewbacca
Hola, Chewbacca.
```

Y eso es una ramificación. Si lo que viene luego del `if` es `true` (es decir, si es cierto), se ejecutará el código entre el `if` y el `end`. Y no se ejecutará si lo que viene luego del `if` es `false` (falso). Más facil imposible.

He indentado el código entre `if` y el `end` solo porque me parece que es más facil leer las ramificaciones de esa manera. Casi todos los programadores lo hacen así, sin importar el lenguaje en que estén programando. No parece ser de mucha ayuda en este ejemploe, pero cuando las cosas se vuelven complejas, es una gran diferencia.

Varias veces nos gustaría que un programa haga alguna cosa si una expresión es `true`, y otra cosa si es que es `false`. Para eso es que existe `else`:

Código:

```
puts 'Soy un adivino. Dime tu nombre:'
name = gets.chomp
if name == 'Chris'
  puts 'Veo grandes cosas en tu futuro.'
else
  puts 'Tu futuro es... ¡Oh! ¡Mira la hora!'
  puts 'Realmente debo irme, ¡lo siento!'
end
```

si respondemos 'Chris':

```
Soy un adivino. Dime tu nombre:
Chris
Veo grandes cosas en tu futuro.
```

Si tratamos con un nombre diferente:

```
Soy un adivino. Dime tu nombre:
Ringo
Tu futuro es... ¡Oh! ¡Mira la hora!
Realmente debo irme, ¡lo siento!
```

Las ramificaciones son como encontrar una bifurcación en el código: ¿Seguimos el camino de las personas cuyo `name == 'Chris'` o el camino de aquellas que no (`else`)?

Y como las ramas de un árbol, puedes tener ramificaciones que a su vez tengan otras ramificaciones:

Código:

```
puts 'Hola, y bienvenido a la clase de 7mo año.'
puts 'Me llamo Mrs. Gabbard. ¿Tú nombre es...?'
nombre = gets.chomp

if nombre == nombre.capitalize
  puts 'Por favor, toma asiento ' + nombre + '.'
else
  puts '¿' + nombre + '? Quieres decir ' + nombre.capitalize + ', ¿cierto?'
  puts '¿No sabes escribir tu propio nombre?'
  respuesta = gets.chomp

  if respuesta.downcase == 'si'
    puts '¡Hum!  Bueno, ¡siéntese!'
  else
    puts '¡SALGA DEL SALON!'
  end
end
```

end
```

*si respondemos el nombre en minúsculas:*

```
Hola, y bienvenido a la clase de 7mo año.
Me llamo Mrs. Gabbard. ¿Tú nombre es...?
chris
¿chris? Quieres decir Chris, ¿cierto?
¿No sabes escribir tu propio nombre?
si
¡Hum!  Bueno, ¡siéntese!
```

y si respodemos el nombre como debe ser:

```
Hola, y bienvenido a la clase de 7mo año.
Me llamo Mrs. Gabbard. ¿Tú nombre es...?
Chris
Por favor, toma asiento Chris.
```

Algunas veces puede ser algo confuso entender donde todos los `if`s, `else`s, y `end`s van. Lo que yo hago es escribir el `end` *al mismo tiempo* que escribo el `if`.
Así es que como se veía al principio el programa de arriba seún lo iba escribiendo:

```
puts 'Hola, y bienvenido a la clase de 7mo año.'
puts 'Me llamo Mrs. Gabbard. ¿Tú nombre es...?'
nombre = gets.chomp

if nombre == nombre.capitalize
else
end
```

Entonces lo llené con *comentarios*, cosas en el código que la computadora ignorará:

```
puts 'Hola, y bienvenido a la clase de 7mo año.'
puts 'Me llamo Mrs. Gabbard. ¿Tú nombre es...?'
nombre = gets.chomp

if nombre == nombre.capitalize
  #  Me trata como una persona normal.
else
  #  Se vuelve loca.
end
```

Todo lo que aparezca luego de un `#` se considera un comentario (a menos, claro, que estés en un texto). Después de eso, reemplacé los comentarios con código funcional. Algunas personas prefieren dejar los comentarios; personalmente, pienso que un código bien escrito normalmente habla por si mismo. Solía utilizar más comentarios, pero mientras más "flúido" me volvía con Ruby menos los utiliza. Actualmente los encuentro algo distrayentes la mayor parte del tiempo. Es una opción personal; tú encontrarás tu propio estilo (siempre en evolución). Así que mi siguiente paso se veía así:

```
puts 'Hola, y bienvenido a la clase de 7mo año.'
puts 'Me llamo Mrs. Gabbard. ¿Tú nombre es...?'
nombre = gets.chomp

if nombre == nombre.capitalize
  puts 'Por favor, toma asiento ' + nombre + '.'
else
  puts '¿' + nombre + '? Quieres decir ' + nombre.capitalize + ', ¿cierto?'
  puts '¿No sabes escribir tu propio nombre?'
  respuesta = gets.chomp

  if respuesta.downcase == 'si'
  else
  end
end
```

De nuevo, escribí los `if`, `else`, y `end` juntos. Realmente me ayuda para saber "donde estoy" en el código. Tambien permite que el trabajo parezca más sencillo ya que me permite enfocarme en una pequeña parte, como en llenar el código entre `if` y el `else`. El otro beneficio de realizarlo de esta manera es que el computador puede entender el programa en cualquier etapa. Cualquiera de las versiones incompletas del programa que te mostrén se podrían ejecutar. No estaban finalizadas, pero eran programas funcionales. De esa manera, podría probarlo mientras lo escribo, lo que ayuda a ver como voy avanzando y donde aún necesito trabajar. Cuando pase todas las pruebas sé que he terminado.

Estos tipos te ayudarán a escribir programas con ramificaciones pero también te ayudarán con otros tipos de control de flujo:
```

Estos tipos te ayudarán a escribir programas con ramificaciones pero también te ayudarán con otros tipos de control de flujo.

Bucles

A veces querrás que tu computadora haga la misma acción una y otra vez —después de todo se supone que en eso son buenas las computadoras—.

Cuando le dices a la computadora que siga repitiendo algo, también debes decirle cuando parar. Las computadoras nunca se aburren asi que si no le indicas cuando hacerlo nunca lo harán. Nos aseguramos que esto no pase al indicarle a la computadora que repita ciertas partes de un programe mientras (`while`) una cierta condición sea cierta. Esto funciona de manera muy similar a como lo hace `if`:

Código:

```
comando = ''

while comando != 'adios'
  puts comando
  comando = gets.chomp
end

puts '¡Vuelve pronto!'
```

Respuesta:

```
¿Hola?
¿Hola?
¡Hola!
¡Hola!
Un gusto conocerlo
Un gusto conocerlo
¡Oh... que amable!
¡Oh... que amable!
adios
¡Vuelve pronto!
```

Y eso es un bucle. (Tal vez te has dado cuenta de la línea vacía que sale al inicio del resultado; viene de ejecutar el primer `puts`, antes del primer `gets`. ¿Cómo cambiarías el programa para eliminar esa primera línea? ¡Pruébalo! ¿Funcionó *exactamente* igual que el programa de arriba, sin contar la primera línea blanca?)

Los bucles nos permiten hacer una serie de cosas interesantes, como estoy seguro te podrás imaginar. Sin embargo, también pueden causar problemas si cometes un error. ¿Qué pasaría si tu computadora se queda atrapado en un bucle infinito? Si piensas que eso te ha ocurrido, solo presiona la tecla `Ctrl` y luego la `C`.

Antes de empezar a jugar con los bucles, aprendadmos un par de cosas que nos permitirán hacer nuestro trabajo más facil.

Un poco de lógica

Echémosle un vistazo a nuestro primer programa de condicionales una vez más. ¿Qué pasaría si mi esposa llega a casa, ve el programa, lo prueba y éste no le dice que *ella* tiene un nombre hermoso. Yo no querría herir sus sentimientos (o dormir en el sillón) así que reescribámoslo:

Código:

```
puts 'Hola, ¿cual es tu nombre?'
nombre = gets.chomp
puts 'Hola, ' + nombre + '.'
if nombre == 'Chris'
  puts '¡Que nombre tan hermoso!'
else
  if nombre == 'Katy'
    puts '¡Que nombre tan hermoso!'
  end
end
```

Respuesta:

```
Hola, ¿cual es tu nombre?
Katy
Hola, Katy.
¡Que nombre tan hermoso!
```

Bueno, funciona... pero no es un programa muy bonito. ¿Por qué no? Bueno, la mejor regla que he aprendido de programación es la regla *DRY*: *Don't Repeat Yourself* ("No te repitas" en español). Probablemente podría escribir un libro pequeño sobre por qué es tan buena esta regla. En nuestro caso, hemos repetido la línea `puts 'Que nombre tan hermoso!'`. ¿Por qué es tan problemático esto? Bueno, que pasaría si cometiera un error al escribir los textos cuando reescribí el programa? ¿Que tal si hubiera querido cambiar de `'hermoso'` a `'bonito'` en ambas líneas?

Soy flojo, ¿recuerdas? Básicamente si quisiera que el programa haga lo mismo cuando reciba `'Chris'` o `'Katy'`, entonces realmente debería hacer exactamente*la misma cosa*:

Código:

```
puts 'Hola, ¿cual es tu nombre?'
nombre = gets.chomp
puts 'Hola, ' + nombre + '.'
if (nombre == 'Chris' or nombre == 'Katy')
  puts '¡Que nombre tan hermoso!'
end
```

Respuesta:

```
Hola, ¿cual es tu nombre?
Katy
Hola, Katy.
¡Que nombre tan hermoso!
```

Mucho mejor. Para hacer que funcione, he utilizado el operador `or`. Los otros *operadores lógicos* son `and` y `not`. Siempre es una buena idea usar los paréntesis cuando trabajamos con éstos. Veamos como funcionan:

Código:

```
soyChris        = true
soyMorado       = false
meGustaLaComida = true
comoRocas       = false

puts (soyChris and meGustaLaComida)
puts (meGustaLaComida and comoRocas)
puts (soyMorado and meGustaLaComida)
puts (soyMorado and comoRocas)
puts
puts (soyChris or meGustaLaComida)
puts (meGustaLaComida or comoRocas)
puts (soyMorado or meGustaLaComida)
puts (soyMorado or comoRocas)
puts
puts (not soyMorado)
puts (not soyChris)
```

Respuesta:

```
true
false
false
false

true
true
true
false

true
false
```

La única de esas sentencias que te podría confundir es el `or`. En español normalmente decimos "uno u otro, pero no los dos". Por ejemplo, tu mamá podría decir "Para postre puedes pedir pie o torta". ¡Ella *no* quiso decir que podrías tener ambos!

Una computadora, por otro lado, usa `or` (o) para decir "uno u otro o los dos" (Otra manera de decirlo es "al menos una de estas debe ser cierta"). Es por eso que las computadoras son más divertidas que mamá.

Algunas cosas por intentar

•*"99 botellas de cerveza en la pared..."* Escribe un programa que imprima la letra del clásico en viajes 99 botellas de cerveza en la pared"

•Escribe un programa de la Abuela Sorda. Lo que sea que le digas a la abuela (lo que sea que escribas), ella deberá responder con HUH?! ¡HABLA MAS FUERTE, HIJO!, `a menos que le grites (escribas todo en mayúsculas) Si le gritas, ella te escuchará (o al menos creerá eso) y te gritará de vuelta NO, ¡NO DESDE 1938!`. Para hacer el programa *realmente* creíble, haz que la abuela grite un año distinto cada vez; tal vez un año aleatorio entre 1930 y 1950. (Esta parte es opcional y sería mucho más facil si lees la sección sobre generadores de números aleatorios de Ruby al final del capítulo métodos) No podrás dejar de hablar con la abuela hasta que le grites `ADIOS`.
***Pista:** ¡No te olvides que* `chomp!` *¡*`'ADIOS'` *con un Enter no es lo mismo que* `'ADIOS'` *sin uno!*
. ***Pista 2:** Trata de pensar que partes del programa deberían suceder una y otra vez. Todas ellas deberían estar en tu bucle*`while`.

• Extiende el programa de la Abuela Sorda. ¿Qué pasaría si la abuela no quiere que te vayas? Cuando le grites `ADIOS`, ella podría pretender que no te escucha. Cambia el programa previo para que tengas que gritar `ADIOS` tres veces *seguidas*. Asegúrate que tu programa: si gritas ADIOS` tres veces, pero no seguidas, deberías seguir hablando con la abuela.

• Años bisiestos. Escribe un programa que pregunte por un año de inicio y uno de fin, y luego imprima (`puts`) todos los años bisiestos que han ocurrido entre ellos (incluyéndolos si también han sido estos años bisiestos). Los años bisiestos son divisibles entre cuatro (como 1984 y 2004). Sin embargo, los años divisibles entre 100 *no* son años bisiestos (como 1800 y 1900) ***a menos*** que sean divisibles por 400 (como 1600 y 2000, los cuales fueron, de hecho, bisiestos). *(Sí, todo es bastante confuso, pero no tanto como tener Enero en medio del invierno, lo cual es algo que a veces sucede)*

Cuando termines con ellos, ¡tómate un descanso! Haz aprendido un montón hasta ahora. ¡Felicitaciones! ¿Estás sorprendido de la cantidad de cosas que le puedes pedir hacer a una computadora? Unos capítulos más y serás capaz de programar cualquier cosas. ¡En serio! Solo mira la cantidad de cosas que puedes hacer ahora que antes cuando no tenías bucles y ramificaciones.

Ahora, aprendamos sobre un nuevo tipo de objeto, que permite tener control de listas de otros objetos: matrices.

Matrices e Iteraciones

Vamos a escribir un programa el cual nos pida digitar tantas palabras como nosotros querramos (una palabra por línea), continuando hasta que oprimamos *Enter* en una línea vacia), el cual luego nos devolverá las palabras en orden alfabetico. OK?

Entonces... nosotros primero vamos; uh... um... hmmm... Bueno, nosotros podríamos; er... um...

Tú sabes, no pienso que podamos hacer esto. Necesitamos una forma de almacenar una cantidad desconocida de palabras y como obtener todas ellas juntas, entonces no se confundan con otras variables. Necesitamos colocarlas en un tipo de lista. Lo que necesitamos son *matrices*.

Una matriz es solo una lista en su computadora. Cada posición en la lista actúa como una variable: puedes ver que objeto en particular apunta a cada posición, y puedes hacer que este apunte a un objeto diferente. Vamos a echar un vistazo a algunas matrices:

```
[]
[5]
['Hola', 'Adiós']

flavor = 'vainilla'           #  Esto no es una matriz, por supuesto...
[89.9, sabor, [true, false]]  #  ...pero esto sí lo es.
```

Entonces primero tenemos una matriz vacía, luego una matriz conteniendo un número simple, luego una matriz que contiene dos textos. Siguiendo, tenemos una simple asignación, luego un array conteniendo tres objetos, de los cuales el último es la matriz `'[true, false]'`. Recuerda, variables no son objetos, entonces nuestra ultima matriz es en realidad un flotante, un *texto*, y una matriz. Aún asi si nosotros hubieramos puestoo `'sabor'` a punto o algo más, esto no hubiera cambiado la matriz.

Para ayudarnos a encontrar un objeto particular en una matriz, cada posición es dada por un índice numérico. Programadores (y, por cierto, la mayoría de los matemáticos) comienzan contando desde cero, por lo que la primera posicion del array es cero. Aquí es como nosotros deberíamos referenciar los objetos en una matriz:

Código:

```
names = ['Ada', 'Belle', 'Chris']

puts names
puts names[0]
puts names[1]
```

```
puts names[2]
puts names[3]  #  Este esta fuera del rango.
```

Resultado:

```
    Ada
    Belle
    Chris
    Ada
    Belle
    Chris
```

Entonces, nosotros vemos que `'puts names'` imprime cada nombre en la matriz `'names'`. Luego usamos `'puts names[0]'` para imprimir el "primer" nombre en la matriz, y `'puts names[1]'` para imprimir el "segundo" ... Estoy seguro que esto parece confuso, pero puedes acostumbrarte a esto. Tienes que realmente solo comenzar pensando que el contador comienza en cero, y dejas de usar palabras como "primero" y "segundo"

Si tienes un conjunto de cinco cursos, no hablas acerca del "primer" curso, hablas acerca del curso cero (y en tu cabeza, estás pensando `'course[0]'`). Tú tienes cinco dedos en tu mano derecha, y sus números son 0, 1, 2, 3 y 4. Mi esposa y yo somos malabaristas. Cuando hacemos malabares con seis objetos, nosotros estamos con los objetos 0-5. Esperamos en los próximos meses, poder manejarnos con el objeto 6(y por lo tanto trataremos de manejarnos con 7 objetos) Tu sabrás que lo has aprendido cuando comiences a usar la palabra "cero". Sí, esto es real; pregunta a cualquier programador o matemático.

Finalmente, tratamos `'puts names[3]'`, solo para ver que podría suceder. ¿Estabas esperando un error? Algunas veces cuando preguntas algo, tu pregunta no tiene sentido (al menos para tu computadora); ahí es cuando obtienes un error.
Algunas veces, en cambio, tu puedes preguntar algo y la respuesta es nada. ¿Qué hay en la posición tres? Nada. ¿Qué hay en `'names[3]'`? `'nil'`: Es la forma Ruby de decir "nada". `'nil'` es un objeto especial que significa "no hay ningún objeto".

Si todo este divertido numerado de posiciones esta molestándote, ¡no te preocupes! También, podemos evitar esto completamente usando varios metodos para matrices, como el que sigue:

El Metodo `'each'`

`'each'` nos permite hacer algo (lo que queramos) a *cada* objeto que apunte. Asi, si queremos decir algo bueno acerca de cada lenguaje en la matriz de abajo, podriamos hacer esto:

Código:

```
lenguajes = ['Inglés', 'Alemán', 'Ruby']

lenguajes.each do |leng|
  puts '¡Me gusta ' + leng + '!'
  puts '¿A ti?'
end

puts '¡Y vamos a escuchar esto para C++!'
puts '...'
```

Resultado:

```
¡Me gusta Inglés!
¿A tí?
¡Me gusta Alemán!
¿A tí?
¡Me gusta Ruby!
¿A tí?
¡Y vamos a escuchar esto para C++!
...
```

¿Qué acaba de ocurrir? Bueno, tenemos permitido ir a través de cada objeto en la matriz sin utilizar ningun número, así esto es definitivamente mejor. Traducido al español diriamos que: Para cada `'each'` objeto en `'lenguajes'`, apunta la variable `'leng'` al objeto y entonces `'do'` (do = hacer) todo lo que te digo, hasta que llegues al `'end'`. (Sólo para que sepas, C++ es otro lenguaje de programación. Este es mucho mas difícil de aprender que Ruby; por lo general, un programa hecho en C++ será muchas veces más extenso que un programa en Ruby que haga la misma funcionalidad)

Tú estarás pensando para ti mismo, "Esto es un montón de bucles como los que hemos aprendido antes". Sí, esto es similar. Una diferencia importante es que el método `'each'` es solo eso: un método. `'while'` y `'end'`; tal como `'do'`, `'if'`, `'else'`, y todas las otras "palabras" no lo son. Ellos son una parte fundamental del lenguaje Ruby, tal como `'='` y los paréntesis son tipos de puntuación como en español.

Pero no `'each'`; `'each'` es solo otro método de la matriz. Métodos como `'each' los cuales 'actuan como' bucles son a menudo llamados *iteradores*.

Algo para saber sobre iteradores es que estos son siempre seguidos por `'do'...'end'`. `'while'` y `'if'` nunca tuvieron un `'do'` cerca de ellos; nosotros solo usamos `'do'` con iteradores

Aquí hay otro pequeño iterador, pero esto no es un método para *matriz...* es un método para *entero*!

Código:

```
3.times do
  puts 'Hip-Hip-Hooray!'
end
```

Resultado:

```
Hip-Hip-Hooray!
Hip-Hip-Hooray!
Hip-Hip-Hooray!
```

Mas métodos para Matrices

Entonces hemos aprendido sobre `'each'`, pero hay muchos métodos más... al menos muchos como métodos para matrices. De hecho, algunos de ellos (como `'length'`, `'reverse'`, `'+'`, y `'*'`) trabajan igual que lo hacen para textos, excepto que ellos operan sobre las posiciones de una matriz y no sobre letras de un texto. Otros, como `'last'` y `'join'`, son específicos de matrices. Aún otros, como `'push'` y `'pop'`, en realidad cambian la matriz. Y así como con métodos para textos, no tienes que recordar todos, puedes recurrir a recordar sobre ellos buscando información.

Primero, vamos a echar un vistazo a `'to_s'` y `'join'`. `'join'` trabaja tal como `'to_s'` lo hace, excepto que este agrega un texto entre los objetos de la matriz. Vamos a mirar un poco:

Código:

```
alimentos = ['artichoke', 'brioche', 'caramel']

puts alimentos
puts
puts alimentos.to_s
puts
puts alimentos.join(', ')
puts
puts alimentos.join('  :)  ') + '  8)'

200.times do
  puts []
end
```

Resultado:

```
artichoke
brioche
caramel

["artichoke", "brioche", "caramel"]

artichoke, brioche, caramel

artichoke  :)  brioche  :)  caramel  8)
```

Como puedes ver, `'puts'` trata la matriz diferente a otros objetos: solo llama `'puts'` sobre cada uno de los objetos en la matriz. Esto es porque `'puts'` con una matriz vacía 200 veces no hace nada; la matriz no apunta a nada; entonces no hay nada para `'puts'`. (Hacer nada 200 veces continúa siendo hacer nada)

Trata de usar `'puts'` en una matriz conteniendo otras matrices; ¿hace lo que esperabas que hiciera?

También, te habrás dado cuenta que dejé fuera el texto vacio cuando quise hacer `'puts'` de una línea en blanco? Esto hace lo mismo.

Ahora vamos a echar un vistazo a `'push'`, `'pop'`, y `'last'`. Los métodos `'push'` y `'pop'` son una suerte de métodos opuestos como lo son

`'+'` y `'-'`. `'push'` agrega un objeto al final de la matriz, y `'pop'` quita el último objeto desde la matriz (y te dice que objeto es). `'last'` es similar a `'pop'` en el sentido que te indica que hay al final de la matriz, excepto que este deja la matriz. De nuevo, `'push'` y `'pop'` en realidad cambian la matriz:

Código:

```
favoritos = []
favoritos.push 'lluvia de rosas'
favoritos.push 'whisky en gatitos'

puts favoritos[0]
puts favoritos.last
puts favoritos.length

puts favoritos.pop
puts favoritos
puts favoritos.length
```

Resultado:

```
lluvia de rosas
whisky en gatitos
2
whisky en gatitos
lluvia de rosas
1
```

Algunas cosas por intentar

- Escribe el programa del cual hablamos al principio de este capítulo. **Consejo:** Hay un hermoso método para matrices el cual te dará la version ordenada de una matriz: `'sort'`. Usalo!
- Escribe el programa sugerido anteriormente sin usar el método `'sort'`. Una gran parte de la programación es la solución de problemas, entonces ¡obtén toda la practica que puedas!
- Reescribe tu programa `Tabla de contenidos` (en el capítulo sobre `'métodos'`). Comienza el programa con una matriz conteniendo toda la información de tu Tabla de Contenidos (capítulo textos, capítulo números, etc.). Entonces imprime la información desde la matriz en una bien formada Tabla de Contenidos.

Hemos aprendido varios métodos diferentes. Ahora es tiempo de aprender como hacerlo por nosotros mismos.

Métodos Propios

Como hemos visto anteriormente, los bucles e iteradores nos permiten hacer lo mismo (o ejecutar el mismo código) una y otra vez. Sin embargo, a veces queremos hacer algo una determinada cantidad de veces pero desde distintos lugares en el programa. Por ejemplo, supongamos que estamos escribiendo un programa que hace un cuestionario para un estudiante de sicología. De los estudiantes de sicología que he conocido y de los cuestionarios que me han dado, esto sería algo similar a:

Código:

```
puts 'Hola, y gracias por tomarse el tiempo para'
puts 'que me ayude con este experimento. Mi experimento'
puts 'tiene que ver con tu gusto acerca de'
puts 'la comida mexicana. Basta con pensar en la comida mexicana'
puts 'y tratar de contestar todas las preguntas con honestidad,'
puts 'ya sea con un "sí" o un "no". Mi experimento'
puts 'no tiene nada que ver con mojar la cama.'
puts

#  Preguntaremos, pero ignoraremos sus respuestas.

buenaRespuesta = false
while (not buenaRespuesta)
  puts '¿Te gusta comer tacos?'
  respuesta = gets.chomp.downcase
  if (respuesta == 'si' or respuesta == 'no')
    buenaRespuesta = true
  else
```

```
    puts 'Por favor, responda "si" o "no".'
  end
end

buenaRespuesta = false
while (not buenaRespuesta)
  puts '¿Te gusta comer burritos?'
  respuesta = gets.chomp.downcase
  if (respuesta == 'si' or respuesta == 'no')
    buenaRespuesta = true
  else
    puts 'Por favor, responda "si" o "no".'
  end
end

#  Presta atención a *esta* respuesta.
buenaRespuesta = false
while (not buenaRespuesta)
  puts '¡Mojas la cama?'
  respuesta = gets.chomp.downcase
  if (respuesta == 'si' or respuesta == 'no')
    buenaRespuesta = true
    if respuesta == 'si'
      mojasLaCama = true
    else
      mojasLaCama = false
    end
  else
    puts 'Por favor, responda "si" o "no".'
  end
end

buenaRespuesta = false
while (not buenaRespuesta)
  puts '¿Te gusta comer chimichangas?'
  respuesta = gets.chomp.downcase
  if (respuesta == 'si' or respuesta == 'no')
    buenaRespuesta = true
  else
    puts 'Por favor, responda "si" o "no".'
  end
end

puts 'Solo una cuantas preguntas más...'

buenaRespuesta = false
while (not buenaRespuesta)
  puts '¿Te gusta comer sopapillas?'
  respuesta = gets.chomp.downcase
  if (respuesta == 'si' or respuesta == 'no')
    buenaRespuesta = true
  else
    puts 'Por favor, responda "si" o "no".'
  end
end

#  Preguntas otras cosas sobre la comida mexicana.

puts
puts 'Interrogatorio:'
puts 'Gracias por tomarse el tiempo para ayudar con'
puts 'este experimento. De hecho, este experimento'
puts 'no tiene nada que ver con la comida mexicana. Es'
puts 'un experimento sobre mojar la cama. La comida mexicana'
puts 'ahí para atraparte con la guardia baja'
puts 'con la esperanza de que respondería más'
puts 'honestamente. Gracias de nuevo.'
puts
puts mojasLaCama
```

Resultado:

```
Hola, y gracias por tomarse el tiempo para
que me ayude con este experimento. Mi experimento
tiene que ver con tu gusto acerca de
la comida mexicana. Basta con pensar en la comida mexicana
y tratar de contestar todas las preguntas con honestidad,
ya sea con un "sí" o un "no". Mi experimento
no tiene nada que ver con mojar la cama.
```

```
no tiene nada que ver con mojar la cama.

¿Te gusta comer tacos?
si
¿Te gusta comer burritos?
si
¡Mojas la cama?
de ninguna manera
Por favor, responda "si" o "no".
¡Mojas la cama?
NO
¿Te gusta comer chimichangas?
si
Solo una cuantas preguntas más...
¿Te gusta comer sopapillas?
si

Interrogatorio:
Gracias por tomarse el tiempo para ayudar con
este experimento. De hecho, este experimento
no tiene nada que ver con la comida mexicana. Es
un experimento sobre mojar la cama. La comida mexicana
ahí para atraparte con la guardia baja
con la esperanza de que respondería más
honestamente. Gracias de nuevo.

false
```

Ese fue un programa bastante largo, con mucha repetición. (Todas las secciones de código sobre las preguntas de comida mexicana eran idénticas, y la pregunta sobre mojar la cama fue la única pregunta algo diferente). La repetición es mala. Sin embargo, no podemos transformar el código en un gran bucle o iterador, porque a veces queremos hacer algo entre las preguntas. En situaciones como estas, es mejor escribir un método. Aquí está como:

```
def decirMu
  puts 'muuuuuu...'
end
```

Uy... nuestro programa no hizo `muuuuuu`. ¿Por qué no?, Porque no le dijimos que lo hiciera. Le dijimos *cómo* hacer `muuuuuu`, pero nunca le dijimos en realidad que lo *haga*. Intentemos otra vez:

Código:

```
def decirMu
  puts 'muuuuuu...'
end

decirMu
decirMu
puts 'coin-coin'
decirMu
decirMu
```

Resultado:

```
muuuuuu...
muuuuuu...
coin-coin
muuuuuu...
muuuuuu...
```

Ah, mucho mejor. (En caso que no hables francés, eso fue un pato francés en medio del programa. En Francia los patos dicen *coin-coin*.)

Por lo tanto, `def`inimos el método de `decirMu`. (Nombres de los métodos, al igual que los nombres de variables, comienzan con letras minúsculas. Hay unas pocas excepciones, tales como `+` o `==`.) ¿Pero los métodos no siempre tienen que estar asociados con objetos?. Bueno, sí, y en este caso (como sucede con `puts` y `gets`), el método sólo está asociado con el objeto que representa la totalidad del programa. En el próximo capítulo veremos cómo agregar métodos a otros objetos. Pero primero ...

Parámetros del Método

Habrás notado que algunos métodos (como `gets`, `to_s`, `reverse`...) solo pueden ser llamados en un objeto. Sin embargo, otros métodos (como `+` , `-` , `puts`...) toman los *parámetros* para decir al objeto que hacer con el método. Por ejemplo, no sólo le diría `5+`, ¿verdad? Le estás pidiendo 5 que agregue, pero no le estás diciendo lo que va a agregar.

Para agregar un parámetro a `decirMu` (digamos, el número de muuuuuu), haríamos lo siguiente:

Código:

```
def decirMu numeroDeMus
  puts 'muuuuuuu...'*numeroDeMus
end

decirMu 3
puts 'oink-oink'
decirMu  #  Esto debería dar un error porque falta el parámetro.
```

Resultado:

```
muuuuuuu...muuuuuuu...muuuuuuu...
oink-oink
in `decirMu': wrong number of arguments (0 for 1) (ArgumentError)
```

`numeroDeMus` es una variable que apunta al parámetro que se le ha pasado, lo voy a decir de nuevo, pero es un poco confuso: `numeroDeMus` es una variable que apunta al parámetro pasado. Por lo tanto, si digito `decirMu 3`, el parámetro es 3, y la variable `numeroDeMus` apunta a 3.

Como puedes ver, el parámetro es ahora *requerido*. Después de todo, lo que `decirMu` hace es multiplicar `'muuuuuu...` ¿pero, si no le dan un parámetro? Tu pobre computadora no tiene ni idea de que hacer.

Si los objetos en Ruby son como los nombres en español, y los métodos son como los verbos, entonces se puede pensar en parámetros como adverbios (como con `decirMu`, donde el parámetro nos dice como se afecta `decirMu`) o a veces como objetos directos (por ejemplo con ´puts´, donde el parámetro es lo que se *muestra*)

Variables Locales

En el siguientre programa hay dos variables:

Código:

```
def duplicaEsto num
  numeroVeces2 = num*2
  puts num.to_s+' el doble es '+numeroVeces2.to_s
end

duplicaEsto 44
```

Resultado:

```
44 el doble es 88
```

Las variables son `num` y `numeroVeces2`. Los dos se encuentran dentro del método `duplicaEsto`. Estas (y todas las variables que hemos visto hasta ahora) son las *variables locales*. Esto significa que ellos viven dentro del método y no pueden salir. Si lo intentas obtendrás un error:

Código:

```
def duplicaEsto num
  numeroVeces2 = num*2
  puts num.to_s+' el doble es '+numeroVeces2.to_s
end

duplicaEsto 44
puts numeroVeces2.to_s
```

Resultado:

```
44 el doble es 88
in `<main>': undefined local variable or method `numeroVeces2' for main:Object (NameError)
```

Variable local no definida ... De hecho, nos *hicieron* definir esa variable local, pero no es local donde traté de usarlo, es local en el método.

Esto podría ser un inconveniente, pero en realidad es bastante bueno. Significa que no tienes acceso a las variables dentro de los métodos y también significa que ellos no tienes acceso a *tus* variables y por lo tanto no pueden meter la pata:

Código:

Código:

```
def pequenaMascota var
  var = nil
  puts '¡HAHA!  ¡He malogrado tu variable!'
end

var = '¡Tu ni siquiera puedes tocar mi variable!'
pequenaMascota var
puts var
```

Resultado:

```
¡HAHA!  ¡He malogrado tu variable!
¡Tu ni siquiera puedes tocar mi variable!
```

En realidad, hay *dos* variables en el programa que se llaman `var`: uno en el interior de `pequenaMascota` y otro fuera de él. Cuando llamamos a `pequenaMascota var`, nosotros en realidad pasamos de un `var` a otro de manera que ambos están apuntando al mismo texto. Luego `pequenaMascota` apuntó a su propio `var` local a `nil`, pero que no hizo nada al `var` que está fuera del método.

Valores de Retorno

Puedes haber notado que algunos métodos envían algo de regreso cuando son llamados. Por ejemplo, `gets` *retorna* una cadena (la cadena que escribiste), y el método `+` en `5+3`, (que es en realidad `5.+(3)`) devuelve `8`. Los métodos aritméticos para los números retornan números y los métodos aritméticos para los textos devuelven textos.

Es importante entender la diferencia entre los métodos que devuelven un valor al método que lo llamó, y lo que el programa muestra en la pantalla como hace `puts`. Ten en cuenta que `5+3` retorna `8`, cosa que *no* hace la salida `8`.

Entonces, ¿qué es lo que *hace* `puts` al retornar? No nos importó antes, pero vamos a ver ahora:

Código:

```
valorRetorno = puts 'Esto es lo que puts retorna:'
puts valorRetorno
```

Resultado:

```
Esto es lo que puts retorna:
cero
```

Así que el primer `puts` retornó `nil`. A pesar de que no lo prueba, el segundo `puts` también lo hizo, `puts` siempre retorna `nil`. Cada método tiene que devolver algo, incluso si es valor es solo `nil`.

Tómate un descanso rápido y escribe un programa para averiguar lo que `decirMu` devuelve.

¿Te sorprendió? Bueno, así es como funciona: el valor devuelto por un método no es más que la última línea del método. En el caso de `decirMu`, esto significa que devuelve `puts "muuuuuu... '* Los numeroDeMus`, lo cual es simplemente `nil` ya que `puts` siempre devuelve `nil`. Si queremos que todos nuestros métodos devuelvan el texto `'submarino amarillo'`, sólo tenemos que ponerla al final de ellos:

Código:

```
def decirMu numeroDeMus
  puts 'muuuuuu...'*numeroDeMus
  'submarino amarillo'
end

x = decirMu 2
puts x
```

Resultado:

```
muuuuuu... muuuuuuu...
submarino amarillo
```

Vamos a tratar con el experimento de sicología otra vez, pero esta vez vamos a escribir un método que haga las preguntas por nosotros. Se tendrá que tomar la pregunta como un parámetro, y devolver `true` si contestan que `sí`, y `false` si responden que `no`. (Aún cuando la mayoría de veces ignoramos la respuesta, sigue siendo una buena idea que nuestro método devuelva la respuesta. De esta manera se puede utilizar para la pregunta de mojar la cama también.) También voy a acortar el saludo y el interrogatorio de manera que sea más fácil de leer:

Código:

```
def preguntar pregunta
  buenarespuesta = false
  while (not buenarespuesta)
    puts pregunta
    reply = gets.chomp.downcase

    if (reply == 'si' or reply == 'no')
      buenarespuesta = true
      if reply == 'si'
        answer = true
      else
        answer = false
      end
    else
      puts 'Por favor, responder "si" o "no".'
    end
  end

  answer  #  Esto es lo que retorna (`true` o `false`).
end

puts 'Hola, y gracias por...'
puts

preguntar '¿Te gusta comer tacos?'      #  Ignoramos la respuesta.
preguntar '¿Te gusta comer burritos?'
mojasLaCama = preguntar '¿Mojas la cama?'  #  Salvamos el valor retornado.
preguntar '¿Te gusta comer chimichangas?'
preguntar '¿Te gusta comer sopapillas?'
preguntar '¿Te gusta comer tamales?'
puts 'Solo unas preguntas mas...'
preguntar '¿Te gusta beber horchata?'
preguntar '¿Te gusta comer flautas?'

puts
puts 'Interrogatorio:'
puts 'Gracias por...'
puts
puts mojasLaCama
```

Resultado:

```
¿Te gusta comer tacos?
si
¿Te gusta comer burritos?
si
¿Mojas la cama?
de ninguna manera
Por favor, responder "si" o "no".
¿Mojas la cama?
NO
¿Te gusta comer chimichangas?
si
¿Te gusta comer sopapillas?
si
¿Te gusta comer tamales?
si
Solo unas preguntas mas...
¿Te gusta beber horchata?
si
¿Te gusta comer flautas?
si

Interrogatorio:
Gracias por...

false
```

No está mal, ¿eh?. Hemos sido capaces de añadir más preguntas (y la adición de preguntas ahora es *fácil*), y nuestro programa es aún más corto!. Es una gran mejora - El sueño de un programador perezoso.

Un ejemplo más grande

Creo que otro método como ejemplo podría ser útil aquí. Llamaremos a este *numeroEnEspanol*. Se llevará un número, como `22`, y devolverá la versión en español de la misma (en este caso el texto de `'veintidós'`) Por ahora, vamos a trabajar solamente con números enteros entre 0 a 100.

*(**NOTA:** Este método utiliza un nuevo truco para retornar en forma temprana en el método usando `return` e introduce un nuevo giro en las bifurcaciones: `elsif`. Debe quedar claro en el contexto de cómo funcionan estos.)*

Código:

```
def numeroEnEspanol numero
  #  Solo estamos considerando los números de 0-100.
  if numero < 0
    return 'Por favor ingrese un numero mayor o igual a cero.'
  end
  if numero > 100
    return 'Por favor ingrese un numero menor o igual a 100'
  end

  numeroDeTexto  = ''  #  Esta es el texto que retorna.

  #  "izquierda" es cuanto del número aún falta escribir.
  #  "escrito" es la parte que estamos escribiendo en estos momentos.
  #  escrito y izquierda... lo captas?  :)
  izquierda  = numero
  escrito = izquierda/100          #  Cuantos cientos faltan escribir?
  izquierda  = izquierda - escrito*100  #  Restar estos cientos.

  if escrito > 0
    return 'cien'
  end

  escrito = izquierda/10          #  Cuantas decenas faltan escribir?
  izquierda  = izquierda - escrito*10  #  Restar las decenas.

  if escrito > 0
    if escrito == 1  #  Ah-ah...
      #  No podemos escribir "diezidos" en lugar de "doce",
      #  hemos realizado una excepción especial .
      if    izquierda == 0
        numeroDeTexto = numeroDeTexto + 'diez'
      elsif izquierda == 1
        numeroDeTexto = numeroDeTexto + 'once'
      elsif izquierda == 2
        numeroDeTexto = numeroDeTexto + 'doce'
      elsif izquierda == 3
        numeroDeTexto = numeroDeTexto + 'trece'
      elsif izquierda == 4
        numeroDeTexto = numeroDeTexto + 'catorce'
      elsif izquierda == 5
        numeroDeTexto = numeroDeTexto + 'quince'
      elsif izquierda == 6
        numeroDeTexto = numeroDeTexto + 'dieciseis'
      elsif izquierda == 7
        numeroDeTexto = numeroDeTexto + 'diecisiete'
      elsif izquierda == 8
        numeroDeTexto = numeroDeTexto + 'dieciocho'
      elsif izquierda == 9
        numeroDeTexto = numeroDeTexto + 'diecinueve'
      end
      #  Desde que ya tomamos el digito faltante,
      #  ya no tenemos nada que escribir.
      izquierda = 0
    elsif escrito == 2
      #  Como no podemos decir "veinteiuno",
      #  tenemos que eliminar la "e" final.
      if izquierda == 0
        numeroDeTexto = numeroDeTexto + 'veinte'
      else
        numeroDeTexto = numeroDeTexto + 'veint'
      end
    elsif escrito == 3
      numeroDeTexto = numeroDeTexto + 'treinta'
    elsif escrito == 4
      numeroDeTexto = numeroDeTexto + 'cuartenta'
    elsif escrito == 5
      numeroDeTexto = numeroDeTexto + 'cincuenta'
```

```
    elsif escrito == 6
      numeroDeTexto = numeroDeTexto + 'sesenta'
    elsif escrito == 7
      numeroDeTexto = numeroDeTexto + 'setenta'
    elsif escrito == 8
      numeroDeTexto = numeroDeTexto + 'ochenta'
    elsif escrito == 9
      numeroDeTexto = numeroDeTexto + 'noventa'
    end

    if izquierda > 0
      numeroDeTexto = numeroDeTexto + 'i'
    end
  end

  escrito = izquierda  #  Cuanto falta para el número?
  izquierda  = 0     #  Restar lo que falta.

  if escrito > 0
    if    escrito == 1
      numeroDeTexto = numeroDeTexto + 'uno'
    elsif escrito == 2
      numeroDeTexto = numeroDeTexto + 'dos'
    elsif escrito == 3
      numeroDeTexto = numeroDeTexto + 'tres'
    elsif escrito == 4
      numeroDeTexto = numeroDeTexto + 'cuatro'
    elsif escrito == 5
      numeroDeTexto = numeroDeTexto + 'cinco'
    elsif escrito == 6
      numeroDeTexto = numeroDeTexto + 'seis'
    elsif escrito == 7
      numeroDeTexto = numeroDeTexto + 'siete'
    elsif escrito == 8
      numeroDeTexto = numeroDeTexto + 'ocho'
    elsif escrito == 9
      numeroDeTexto = numeroDeTexto + 'nueve'
    end
  end

  if numeroDeTexto == ''
    #  La unica forma que "numeroDeTexto" este vacio es que
    #  "numero" sea 0.
    return 'cero'
  end

  #  Si numeroDeTexto os hasta hasta aqui es que tenemos un numero
  #  entre 0 y 100, por lo que debemos retornar "numeroDeTexto".
  numeroDeTexto
end

puts numeroEnEspanol(  0)
puts numeroEnEspanol(  9)
puts numeroEnEspanol( 10)
puts numeroEnEspanol( 21)
puts numeroEnEspanol( 17)
puts numeroEnEspanol( 32)
puts numeroEnEspanol( 88)
puts numeroEnEspanol( 99)
puts numeroEnEspanol(100)
```

Resultado:

```
cero
nueve
diez
veintiuno
diecisiete
treintaidos
ochentaiocho
noventainueve
cien
```

Bueno, sin duda hay algunas cosas acerca de este programa que no me gustan. En primer lugar hay demasiada repetición. En segundo lugar, no se ocupa de los números mayores de 100. En tercer lugar, hay muchos casos especiales, demasiados `return`s. Veamos ahora algunas matrices y tratar de limpiar un poco:

Código:

```
def numeroEnEspanol numero
  if numero < 0  #  No numeros negativos
    return 'Por favor ingresar un numero que nos ea negativo.'
  end
  if numero == 0
    return 'cero'
  end

  #  No mas casos especiales!  No mas returns!

  numeroDeTexto = ''  #  Esta es el texto que se retornara.

  primeraPosicion = ['uno',     'dos',       'tres',    'cuatro',     'cinco',
                'seis',     'siete',     'ocho',    'nueve']
  segundaPosicion = ['diez',     'veinte',    'treinta',   'cuarenta',    'cincuenta',
                'sesenta',   'setenta',   'ochenta',   'noventa']
  entre11Y19 = ['once',  'doce',    'trece', 'catorce', 'quince',
                'dieciseis', 'diecisiete', 'dieciocho', 'diecinueve']
  entre21Y29 = ['veintiuno',  'veintidos',    'veintitres', 'veinticuatro', 'veinticinco',
                'veintiseis', 'veintisiete', 'veintiiocho', 'veintinueve']

  #  "izquierda" es cuanto del numero aun falta escribir
  #  "escrito" es la parte que estamos escribiendo en este momento.
  #  escrito y izquierda... lo captas?  :)
  izquierda  = numero
  escrito = izquierda/100          #  Cuantos cientos faltan escribir?
  izquierda  = izquierda - escrito*100  #  La esta de estos cientos

  if escrito > 0
    #  Ahora hacemos un pequeño truco:
    cientos  = numeroEnEspanol escrito
    numeroDeTexto = numeroDeTexto + cientos + ' ciento'

    # Eso se llama "recursividad". Entonces, ¿Qué acabo de hacer?
    # Le dije a este método para llamarse a sí mismo, pero con "escrito" en vez de
    # "numero". Recuerde que "escrito" es (por el momento) el número de
    # cientos que tenemos que escribir. Después añadimos "hundred" de "numeroDeTexto",
    # añadimos la cadena 'cien' después de él. Así, por ejemplo, si
    # se llamó originalmente numeroEnEspanol con el 1999 (por lo que "numero" = 1999),
    # y luego en este momento "escrito" sería 19, y la "izquierda" sería 99.
    # La más perezoso que se puede hacer en ese momento es que numeroEnEspanol
    # escriba 'diecinueve' para nosotros, entonces escribimos 'cien',
    # y luego el resto de numeroEnEspanol escribe "noventa y nueve".

    if izquierda > 0
      #  So escribir  'dos ciencincuentaiuno'...
      numeroDeTexto = numeroDeTexto + ' '
    end
  end

  escrito = izquierda/10          #  Cuantas decenas faltan escribir?
  izquierda  = izquierda - escrito*10  #  Resta de estas decenas.

  if escrito > 0
    if ((escrito == 1) and (izquierda > 0))
      #  No podemos escribir "diez-dos" instead of "doce",
      #  hacemos una excepción especial .
      numeroDeTexto = numeroDeTexto + entre11Y19[izquierda-1]
      #  Es "-1" porque entre11Y19[3] ess 'catorce', no 'trece'.

      #  Ya que tomamos el digito que nos faltaba,
      #  no tenemos nada mas que escribir.
      izquierda = 0
    elsif ((escrito ==2 and (izquierda>0)))
      #  Similar para los veintipico
      numeroDeTexto = numeroDeTexto + entre21Y29[izquierda-1]
      izquierda = 0
    else
      numeroDeTexto = numeroDeTexto + segundaPosicion[escrito-1]
      #  Es "-1" porque segundaPosicion[3] es 'cuarenta', no 'treinta'.
    end

    if izquierda > 0
      #  No escribiremos 'sesentacuatro'...
      numeroDeTexto = numeroDeTexto + 'i'
    end
  end
```

```
  end

  escrito = izquierda  #  Cuantos faltan?
  izquierda  = 0     #  Restar lo que falta.

  if escrito > 0
    numeroDeTexto = numeroDeTexto + primeraPosicion[escrito-1]
    #  Es "-1" porque primeraPosicion[3] es 'cuatro', no 'tres'.
  end

  #  Ahora solo retornamos "numeroDeTexto"...
  numeroDeTexto
end

puts numeroEnEspanol(  0)
puts numeroEnEspanol(  9)
puts numeroEnEspanol( 10)
puts numeroEnEspanol( 11)
puts numeroEnEspanol( 17)
puts numeroEnEspanol( 32)
puts numeroEnEspanol( 88)
puts numeroEnEspanol( 99)
puts numeroEnEspanol(100)
puts numeroEnEspanol(101)
puts numeroEnEspanol(234)
puts numeroEnEspanol(3211)
puts numeroEnEspanol(999999)
puts numeroEnEspanol(1000000000000)
```

Resultado:

```
cero
nueve
diez
once
diecisiete
treintaidos
ochentaiocho
noventainueve
uno ciento
uno ciento uno
dos ciento treintaicuatro
treintaidos ciento once
noventainueve ciento noventainueve ciento noventainueve
uno ciento ciento ciento ciento ciento ciento
```

Ahhhh .... Eso está mucho, mucho mejor. El programa es bastante denso, por lo que puse varioos comentarios. También funciona con números grandes ... aunque no tan bien como cabría esperar. Por ejemplo, creo que ' un millón de millones ' sería un valor de retorno más agradable para que el último número. De hecho, puedes hacer eso en este momento ...

Algunas cosas por intentar

- Ampliar al `numeroEnEspanol` . En primer lugar, poner en miles. Por lo tanto, debe devolver `un mil` en lugar de `diez cien` y `diez mil` en lugar de `cien cien` .

- Ampliar al `numeroEnEspanol` un poco más. Ahora ponga en millones, para que pueda obtener `un millón` en lugar de `un mil mil`. A continuación, pruebe a añadir miles de millones y billones. ¿Qué tan alto puede llegar?

- ¿Qué hay de `tiempoDeMatrimonio`? Deberá trabajar casi como `numeroEnEspanol`, excepto que se debe insertar la palabra "y" por todo el lugar, volviendo las cosas como `` `mil movecientos setenta y dos' ``, o como se supone deben verse en las invitaciones de la boda. Te daría más ejemplos, pero yo no lo entiendo completamente. Es posible que necesite ponerse en contacto con un coordinador de bodas para que le ayude.

- Las *"Noventa y nueve botellas de cerveza ..."* Usando numeroEnEspanol y el viejo programa, escribir las letras de esta canción en forma *correcta* esta vez. Penalizar a tu computador: hacer que se inicie en 9999. (No elegir un número demasiado grande, ya que al escribir todo eso en la pantalla de tu computador toma un buen tiempo. Cien mil botellas de cerveza lleva tiempo;. Y si tienes que elegir un millón, te estarás castigando tú mismo!

¡Felicitaciones! En este punto, usted es un ¡programador de verdad! Usted ha aprendido todo lo que necesita para construir grandes programas desde cero. Si tienes ideas para los programas que te gustaría escribir para ti mismo, ¡dame una!

Por supuesto, construir todo desde cero puede ser un proceso bastante lento. ¿Por qué gastar tiempo escribiendo código que alguien más ya ha escrito? ¿Quieres que tu programa envie un correo electrónico? ¿Te gustaría guardar y cargar archivos en tu computadora? ¿Qué hay de la generación de páginas web para ver un tutorial en donde los ejemplos de código son automáticamente ejecutados? ;) Ruby tiene muchos tipos de objetos que podemos utilizar para ayudarnos a escribir mejores programas y con mayor rapidez.

Clases

Hasta ahora hemos visto diferentes tipos o *clases* de objetos: textos, enteros, flotantes, matrices, y algunos objetos especiales (`true`, `false` y `nil`) de los cuales hablaremos más tarde. En Ruby, estas clases están siempre en mayúsculas: `String`, `Integer`, `Float`, `Array` ... etc. En general, si queremos crear un nuevo objeto de una cierta clase, usamos `new`:

Código:

```
a = Array.new  + [12345]  #  Array  agregado.
b = String.new + 'hello'  #  String agregado.
c = Time.new

puts 'a = '+a.to_s
puts 'b = '+b.to_s
puts 'c = '+c.to_s
```

Resultado:

```
a = [12345]
b = hello
c = 2012-06-01 12:52:06 -0500
```

Porque podemos crear matrices y textos usando `[...]` y `' ... '` respectivamente es que rara vez los creamos usando `new`. (Aunque no es muy evidente en el ejemplo anterior, `String.new` crea una cadena vacía y `Array.new` crea una matriz vacía) También, los números son excepciones especiales: no se puede crear un número entero con `Integer.new` . Sólo tienes que escribir el número entero.

La clase `Time`

Entonces, ¿cuál es la historia con la clase `Time`?. Los objetos `Time` representan momentos en el tiempo. Usted puede sumar (o restar) números a (o desde) para obtener los nuevos tiempos: la adición de `1.5` a un tiempo hace un tiempo nuevo segundo y medio más tarde:

Código:

```
hora  = Time.new   #  El momento que se ejecutó esta código
hora2 = hora + 60  #  Un minuto más tarde.

puts hora
puts hora2
```

Resultado:

```
2012-06-01 12:55:49 -0500
2012-06-01 12:56:49 -0500
```

También puedes obtener el tiempo para un momento específico utilizando `Time.mktime`:

Código:

```
puts Time.mktime(2000, 1, 1)          #  Y2K.
puts Time.mktime(1976, 8, 3, 10, 11)  #  Cuando nació Chris.
```

Resultado:

```
2000-01-01 00:00:00 -0500
1976-08-03 10:11:00 -0500
```

Nota: yo nací en el horario de verano del Pacífico (PDT). Cuando se presentó el problema del año 2000, sin embargo, era hora estándar del Pacífico (PST), por lo menos para nosotros los de la Costa Oeste. Los paréntesis son para agrupar los parámetros a `mktime`. Los parámetros adicionales hacen más preciso el tiempo obtenido.

Puedes comparar los tiempos usando los métodos de comparación (un tiempo anterior es *menos que* un tiempo posterior), y si se resta un tiempo de otro obtendrás el número de segundos entre ellos. Juega un poco con él!

tiempo de otro obtendrás el número de segundos entre ellos. Juega un poco con él.

Algunas cosas para probar

- Mil millones de segundos ... Descubre el segundo exacto en que naciste (si puedes). Averigua cuando tendrás (o tal vez cuando tuviste?) mil millones de segundos de edad, luego marcalo en tu calendario.
- ¡Feliz cumpleaños! Pregunta en qué año nació una persona, luego el mes, y luego el día. Calcula la edad que tiene y dale una gran ´¡NALGADA!´ por cada cumpleaños que ha tenido.

La clase`Hash`

Otra clase muy útil es `Hash`. Los valores hash son muy parecidos a las matrices: tienen un montón de ranuras que pueden apuntar a objetos diferentes. Sin embargo, en una matriz, las ranuras están alineados en una fila y cada uno está numerado (empezando de cero). En un `hash`, las ranuras no están en una fila (que es sólo un tipo de mezcla), y se puede utilizar *cualquier* objeto para hacer referencia a un espacio no sólo un número. Es bueno utilizar hashes cuando se tiene un montón de cosas que no se desea perder de vista, pero que en realidad no encajan en una lista ordenada. Por ejemplo los colores que yo uso para las diferentes partes del código con las que he creado este tutorial:

Código:

```
colorArray = []  #  igual a Array.new
colorHash  = {}  #  igual a Hash.new

colorArray[0]         = '#FF0000'
colorArray[1]         = '#008000'
colorArray[2]         = '#0000FF'
colorHash['textos ']  = '#FF0000'   # rojo
colorHash['numeros']  = '#008000'   # verde
colorHash['claves ']  = '#0000FF'   # azul

colorArray.each do |color|
  puts color
end
colorHash.each do |tipoCodigo, color|
  puts tipoCodigo + ' :  ' + color
end
```

Resultado:

```
#FF0000
#008000
#0000FF
textos  :  #FF0000
numeros :  #008000
claves  :  #0000FF
```

Si utilizo una matriz, tengo que recordar que la ranura `0` es para las textos, la ranura `1` es para los números, etc, pero si puedo usar un hash, ¡es muy fácil! Ranura `' textos '` mantiene el color de las cadenas, por supuesto. No hay nada que recordar. Puedes haber notado que cuando se utiliza `each` los objetos en el hash no vienen en el mismo orden en que los pusiste adentro. (Al menos no lo hacían cuando escribí esto. Pueden que lo hagan ahora... nunca se sabe con los hashes). Matrices son para mantener las cosas en orden, no hashes.

Aunque la gente suele utilizar textos para nombrar las ranuras de un hash, se puede utilizar cualquier tipo de objeto, incluso las matrices y los hashes de otros (aunque yo no puedo pensar en por qué querría hacer esto ...):

```
weirdHash = Hash.new

weirdHash[12] = 'monos'
weirdHash[[]] = 'el vacío'
weirdHash[Time.new] = 'ningun momento como el actual'
```

Hashes y matrices son buenos para cosas diferentes, depende de ti decidir cuál es el mejor para un problema particular.

Extendiendo las clases

Al final del último capítulo escribiste un método para decir la frase en español de un número entero. No era un método de número entero, sin embargo; fue sólo un método genérico de un "programa". ¿No sería agradable si pudieras escribir algo como `22.to_esp` en lugar de `numeroEspanol 22`?. He aquí cómo harías eso:

Código:

```
class Integer
  def to_esp
    if self == 5
      espanol = 'cinco'
    else
      espanol = 'cincuenta y ocho'
    end

    espanol
  end
end

#  Mejor probarlo en un par de números...
puts 5.to_esp
puts 58.to_esp
```

Resultado:

```
cinco
cincuenta y ocho
```

Bueno, lo he probado y parece que funciona. ;)

Por lo tanto, definimos un método del número entero ingresando en la clase `Integer`, definiendo el método allí y saliendo luego. Ahora todos los números enteros tienen este (algo incompleto) método. De hecho, si no te gusta la forma en que el método `to_s` trabaja, lo podrías definir en gran parte de la misma manera ... pero ¡yo no lo recomiendo! Lo mejor es dejar los viejos métodos y hacer otros nuevos cuando se quiere hacer algo nuevo.

Así que ... ¿confundido? Permiteme explayarme más en ese último programa. Hasta ahora, cada vez que ejecutabas cualquier código o cualquier otro método definido se hacía por omisión en el "programa" del objeto. En nuestro último programa, dejamos el objeto por primera vez y se fuimos a la clase `Integer`. Se definió un método allí (lo que lo convierte en un método de número entero) y que todos los números enteros pueden utilizar. Dentro de ese método se utiliza `self` para referirse al objeto (el entero) utilizando el método.

Creando clases

Hemos visto diferentes clases de objetos. Sin embargo, es fácil llegar a los tipos de objetos que Ruby no tiene. Por suerte, la creación de una nueva clase es tan fácil como la ampliación de una antigua. Digamos que hemos querido hacer algunos dados en Ruby. Así es como podemos hacer la clase Dado:

Código:

```
class Dado

  def roll
    1 + rand(6)
  end

end

#  Vamos a crear un par de dados...
dados = [Dado.new, Dado.new]

#  ...y hacerlos rodar.
dados.each do |dado|
  puts dado.roll
end
```

Resultado:

```
3
4
```

(Si se ha saltado la sección de números aleatorios, `rand(6)` sólo da un número aleatorio entre 0 y 5 .) ¡Y eso es todo! Nuestro propios objetos. Tira los dados un par de veces (con el botón de recarga) y mira lo que aparece.

Podemos definir todo tipo de métodos para los objetos ... pero hay algo que falta. Trabajar con estos objetos se parece mucho a la programación antes de aprender acerca de las variables. ¡Mira los dados!, por ejemplo. Podemos rodarlos y cada vez nos dan números diferentes. Pero si quería guardar alguno de estos números habría que crear una variable que apunte al número. Parece que cualquier dado

decente debe ser capaz de *tener* un número, y que rodando el dado nuevamente el número debe cambiar. Si hacemos un seguimiento del dado debemos tener también un registro del número mostrado.

Sin embargo, si tratamos de guardar el número que salió en una variable (local) `roll` habrá desaparecido tan pronto como `roll` haya terminado. Tenemos que guardar el número en un tipo diferente de variable.

Variables de instancia

Normalmente cuando queremos hablar de un texto, nos limitaremos a llamar a un *texto*. Sin embargo, también podría llamar un *objeto de texto*. Sin embargo, los programadores podrían llamar *una instancia de la clase `String`*, pero esto es sólo una manera de representar a *texto*. Una *instancia* de una clase es sólo un objeto de esa clase.

Por lo tanto las variables de instancia son variables de un objeto. Las variables locales de un método viven hasta que el método ha terminado. Las variables de instancia de un objeto, por otro lado, duran tanto como el objeto. Para diferenciar las variables de instancia de las variables locales, ellos tienen `@` delante de sus nombres:

Código:

```
class Dado

  def rodar
    @numeroMostrado = 1 + rand(6)
  end

  def mostrar
    @numeroMostrado
  end

end

dado = Dado.new
dado.rodar
puts dado.mostrar
puts dado.mostrar
dado.rodar
puts dado.mostrar
puts dado.mostrar
```

Resultado:

```
1
1
5
5
```

¡Muy bien!. `rodar` rueda el dado y `mostrar` nos dice el número que muestra. Sin embargo, que pasa si tratamos de mostrar lo que existía antes de rodar el dado (antes de que lo hayamos guardado en `@numeroMostrado`)

Código:

```
class Dado

  def rodar
    @nummeroMostrado = 1 + rand(6)
  end

  def mostrar
    @numeroMostrado
  end

end

# Ya que no voy a usar esta dado de nuevo,
# No es necesario guardarlo en una variable.
puts Dado.new.mostrar
```

Resultado:

```
nil
```

Hmmm ... bueno, al menos no nos muestra un error. Aún así, no tiene mucho sentido para un dado "no rodado" mostrar `nil` o lo que sea que

se supone que significa. Sería bueno si pudiéramos dar un valor cuando nuestro objeto Dado es creado. Para esto esta `initialize`:

Código:

```
class Dado

  def initialize
    # Voy a tirar el dado, a pesar de que
    # podria hacer otra cosa si quisieramos
      rodar
  end

  def rodar
    @numeroMostrado = 1 + rand(6)
  end

  def mostrar
    @numeroMostrado
  end

end

puts Dado.new.mostrar
```

Resultado:

```
6
```

Cuando se crea un objeto, su método `initialize` (si se ha definido uno) siempre es llamado. Nuestros dados son casi perfectos. La única cosa que podría hacer falta es una manera de decirle que lado del dado debe mostrar... ¿Por qué no escribir un método `timo` que hace justamente eso! Vuelve cuando hayas terminado (y que haya funcionado por supuesto). Asegúrese de que nadie pueda obtener un `7` con el dado!

Hay varios temas interesante que apenas hemos revisado. Es difícil, sin embargo te voy a dar otro ejemplo más interesante. Digamos que queremos hacer una mascota virtual sencilla, un dragón bebé. Como la mayoría de los bebés debe ser capaz de comer, dormir y hacer sus necesidades, lo que significa que tendremos que ser capaces de darle de comer, de ponerlo en la cama y llevarlo a pasear. Internamente, nuestro dragón tendrá que realizar una verificación de si tiene hambre, está cansado o tiene que ir al baño pero no vamos a ser capaces de ver su estado cuando nos relacionamos con nuestro dragón al igual que no se puede preguntar a un bebé humano, "¿Tienes hambre?". También vamos a añadir algunas otras maneras divertidas de interactuar con nuestro dragón bebé y cuando nace le daremos un nombre. (Lo que sea que pasen al método `new` al método `initialize`). Muy bien, continuemos:

Código:

```
class Dragon

  def initialize nombre
    @nombre = nombre
    @dormido = false
    @panzaLlena     = 10  #  Esta lleno.
    @intestinoLleno  =  0  #  No necesita ir.

    puts @nombre + ' nace.'
  end

  def alimentar
    puts 'Alimentas a ' + @nombre + '.'
    @panzaLlena = 10
    pasoDelTiempo
  end

  def caminar
    puts 'Haces caminar a ' + @nombre + '.'
    @intestinoLleno  = 0
    pasoDelTiempo
  end

  def dormir
    puts 'Colocas a ' + @nombre + ' en la cama.'
    @dormido = true
    3.times do
      if @dormido
        pasoDelTiempo
      end
      if @dormido
```

```
      puts @nombre + ' ronca, llenando el cuarto con humo.'
    end
  end
  if @dormido
    @dormido = false
    puts @nombre + ' despierta lentamente.'
  end
end

def lanzar
  puts 'Lanzas a ' + @nombre + ' en el aire.'
  puts 'Sonrie, sus cejas se mueven.'
  pasoDelTiempo
end

def acunar
  puts 'Acunas a ' + @nombre + ' suavemente.'
  @dormido = true
  puts 'Rapidamente se queda dormido...'
  pasoDelTiempo
  if @dormido
    @dormido = false
    puts '...pero despierta cuando te detienes.'
  end
end

private

#  "private" significa que los metodos defindos aqui son
#  metodos internos al objeto.  (Puedes alimentar a
#  tu dragon, pero no puedes preguntar si esta hambriento.)

def hambriento?
  #  Los nombres de los metodos pueden terminar en "?".
  #  Generalmente, hacemos esto si el método debe
  #  devolver verdadero o falso, como esto:
  @panzaLlena <= 2
end

def ganas?
  @intestinoLleno >= 8
end

def pasoDelTiempo
  if @panzaLlena > 0
    #  Mueve el alimento del vientre al intestino.
    @panzaLlena      = @panzaLlena      - 1
    @intestinoLleno  = @intestinoLleno  + 1
  else  #  Nuestro dragon esta hambriento!
    if @dormido
      @dormido = false
      puts '¡Se despierta de repente!'
    end
    puts '¡' + @nombre + ' esta hambriento!  En su desperacion, ¡te COMIO!'
    exit  #  Sale del programa.
  end

  if @intestinoLleno  >= 10
    @intestinoLleno  = 0
    puts '¡Uy!  ' + @nombre + ' tuvo un accidente...'
  end

  if hambriento?
    if @dormido
      @dormido = false
      puts '¡Se despierta de repente!'
    end
    puts 'El estomago de ' + @nombre + 'retumba...'
  end

  if ganas?
    if @dormido
      @dormido = false
      puts 'Se despierta de repente!'
    end
    puts @nombre + ' hace la danza del baño...'
  end
end
```

```
end

mascota = Dragon.new 'Norbert'
mascota.alimentar
mascota.lanzar
mascota.caminar
mascota.dormir
mascota.acunar
mascota.dormir
mascota.dormir
mascota.dormir
mascota.dormir
```

Resultado:

```
Norbert nace.
Alimentas a Norbert.
Lanzas a Norbert en el aire.
Sonrie, sus cejas se mueven.
Haces caminar a Norbert.
Colocas a Norbert en la cama.
Norbert ronca, llenando el cuarto con humo.
Norbert ronca, llenando el cuarto con humo.
Norbert ronca, llenando el cuarto con humo.
Norbert despierta lentamente.
Acunas a Norbert suavemente.
Rapidamente se queda dormido...
...pero despierta cuando te detienes.
Colocas a Norbert en la cama.
¡Se despierta de repente!
El estomago de Norbertretumba...
Colocas a Norbert en la cama.
¡Se despierta de repente!
El estomago de Norbertretumba...
Colocas a Norbert en la cama.
¡Se despierta de repente!
El estomago de Norbertretumba...
Norbert hace la danza del baño...
Colocas a Norbert en la cama.
¡Se despierta de repente!
¡Norbert esta hambriento!  En su desperacion, ¡te COMIO!
```

¡Ouau! Por supuesto, sería mejor si fuese un programa interactivo, pero puedes modificarlo. Yo sólo estaba tratando de mostrarte las partes relacionadas directamente con la creación de una nueva clase de dragón.

Vimos algunas cosas nuevas en ese ejemplo. La primera es simple: `exit` termina el programa en ese mismo momento. La segunda es la palabra `private`, que pusimos en la definición de nuestra clase. Podría haberlos mantenido afuera, pero yo quería reforzar la idea de que hay ciertos métodos que hacen cosas que puede hacer un dragón y otros que simplemente se establecen dentro el dragón. Usted puede pensar en ellos como "bajo el capó": a menos que seas un mecánico de automóviles todo lo que necesitas saber es el pedal del acelerador, el pedal del freno y el volante. Un programador puede llamar a esto *interfaz pública* del automóvil. Sin embargo, cómo saber cuando la bolsa de aire se activará es algo interno del automóvil, el usuario típico (el conductor) no tiene por qué saberlo.

En realidad, para un ejemplo más concreto en ese sentido vamos a hablar sobre cómo se podría representar un coche en un videojuego (que pasa a ser mi línea de trabajo). En primer lugar, tienes que decidir como deseas que tu interfaz pública se parezca, es decir, los métodos de la gente debería ser capaz de llamar por cada uno de los objetos de su coche. Bueno, tienes que ser capaz de empujar el pedal del acelerador y el pedal de freno, pero que también tienes que ser capaz de especificar lo fuerte que estás empujando el pedal. (Hay una gran diferencia entre pisar y golpear) También tendrías que ser capaz de dirigir el timón, y otra vez, tendrías que ser capaz de decir lo mucho que estás moviendo la rueda del timón. Supongo que se podría ir más allá y añadir un embrague, direccionales, lanzacohetes, cámara de postcombustión, condensador de flujo, etc .. depende del tipo de juego que estás haciendo.

Internamente al objeto coche, sin embargo, tendríamos que hacer mucho más cosas; otras cosas que necesita un coche son una velocidad, una dirección y una posición (que son las cosas más básicas). Estos atributos se modificarían pulsando sobre el gas o los pedales de freno y girando la rueda, pero el usuario no sería capaz de establecer la posición directamente (que sería como una deformación). Usted también podría patinar o dañar si se ha volcado y así sucesivamente. Todo esto sería interno a su objeto coche.

Algunas cosas por intentar

- Hacer una clase `arbolNaranja`. Deberá tener un método `altura` que devuelve su altura y un método `paso365Dias` que cuando se le llama aumenta la edad del árbol en un año. Cada año crece el árbol más alto (mucho más de lo que piensas que un naranjo debe crecer en un año), y después de un cierto número de años (de nuevo, tu llamada) el árbol debe morir. En los primeros años el árbol no debe

producir fruta, pero después de un tiempo debería hacerlo, y supongo que los árboles más viejos producen más cada año que los árboles más jóvenes ... lo que pienses que tiene más sentido. Y, por supuesto, tú deberás ser capaz de `contarNaranjas` (que devuelve el número de naranjas en el árbol), y `tomarUnaNaranja` (que reduce la `@numeroNaranjas`en uno y devuelve un texto que te dice cómo la naranja era deliciosa, o de lo contrario sólo te dice que no hay naranjas más para elegir este año) Asegúrate que las naranjas que no se recogen en un año se caen antes del próximo año.

- Escribir un programa para que pueda interactuar con tu bebé dragón. Deberás ser capaz de introducir comandos como `alimentar` y `caminar`, y hacer que esos métodos se llamen en tu dragón. Por supuesto, ya que lo que estás ingresando son sólo textos, tendrás que tener algún tipo de *método de envío*, donde el programa revise los textos que se han ingresado y luego llama al método adecuado.

¡Y eso es casi todo lo que hay que hacer! Pero espera un segundo ... Yo no he hablado de ninguna de esas clases para hacer cosas como enviar un correo electrónico o guardar y cargar archivos en el cumputador, o cómo crear ventanas y botones, o los mundos en 3D, ¡ni nada! Bueno, hay *tantas* clases que pueden utilizar que no es posible mostrar a todos, ¡yo no conozco la mayoría de ellos! Lo *que* puedo decir es que para saber más sobre ellos tienen que saber acerca de los que deseas que el programa haga. Antes de terminar hay una característica más de Ruby que deberás conocer, algo que la mayoría de lenguajes de programación no tiene pero sin las cuales simplemente no podría vivir: bloques y procedimientos.

Bloques y Procedimientos

Esta es definitivamente una de las mejores herramientas de Ruby. Algunos lenguajes tienen esta herramienta, pienso que la llamaran de otra forma (como *closures*), pero la mayoría de los más populares no lo hacen, una pena.

Entonces ¿qué es esto que es tan bueno? Esto tiene la habilidad de tomar un *bloque* de código (código entre **do** y **end**) y encapsularlo dentro de un objeto (llamado *proc*) y guardarlo en una variable o pasarlo a un método, y ejecutar el código del bloque donde te guste (más de una vez, si quieres) Entonces esto es como un tipo método excepto que no está dentro de un objeto (este bloque *es* un objeto), y puedes almacenarlo o pasarlo como cualquier otro objeto. Es hora de un ejemplo:

Código:

```
toast = Proc.new do
  puts '¡Aplausos!'
end

toast.call
toast.call
toast.call
```

Resultado:

```
¡Aplausos!
¡Aplausos!
¡Aplausos!
```

Entonces creé un proc (el cual pienzo debería ser pronunciado como "procedimiento") que contiene un bloque de código, y llamé (*call*ed) el proc tres veces. Como puedes ver, esto es como un método.

En realidad, son más parecidos a los métodos que te he mostrado, porque los bloques pueden tomar parámetros:

Código:

```
teGusta = Proc.new do |algoRico|
  puts '¡Me gusta *realmente* el '+algoRico+'!'
end

teGusta.call 'chocolate'
teGusta.call 'ruby'
```

Resultado:

```
¡Me gusta *realmente* el chocolate!
¡Me gusta *realmente* el ruby!
```

Muy bien, entonces hemos visto que son los bloques y procs, y como usarlos, pero ¿cuál es el punto? ¿Porqué no utilizar simplemente métodos? Bueno, esto es porque hay más cosas que no podemos hacer con métodos. En particular, no puedes pasar métodos a otros métodos (pero puedes pasar procs dentro de métodos), y métodos no pueden retornar otros métodos (pero ellos pueden retornar procs). Esto es

simplemente porque procs son objetos; los métodos no son objetos

(De hecho, ¿es algo familiar para tí? Sí, tu has visto bloques antes... cuando aprendiste sobre iteradores. Pero vamos a hablar un poco más acerca de esto en breve)

Métodos que reciben Procedimientos

Cuando pasamos un proc en un un método podemos controlar cómo o cuántas veces llamamos el proc. Por ejemplo, vamos a decir que queremos hacer antes y después de cierto código que se esta ejecutando:

Código:

```
def hacerAlgoImportante unProc
  puts '¡Todo el mundo DETENGANSE!  Tengo algo que hacer...'
  unProc.call
  puts 'A todos: Está hecho.  Continuen con lo que estaban haciendo.'
end

decirHola = Proc.new do
  puts 'hola'
end

decirAdios = Proc.new do
  puts 'adios'
end

hacerAlgoImportante decirHola
hacerAlgoImportante decirAdios
```

Resultado:

```
¡Todo el mundo DETENGANSE!  Tengo algo que hacer...
hola
A todos: Está hecho.  Continuen con lo que estaban haciendo.'
¡Todo el mundo DETENGANSE!  Tengo algo que hacer...
adios
A todos: Está hecho.  Continuen con lo que estaban haciendo.'
```

Quizá esto no parezca muy fabuloso... pero lo es. :-) Es común en programacion tener requerimientos estrictos acerca de que debe ser hecho y cuando. Si quieres salvar un archivo, por ejemplo, tienes que abrir el archivo, escribir la informacion que quieres que contenga y luego cerrar el archivo. El olvido de cerrar el archivo puede traer malas consecuencias. Pero cada vez que quieras salvar un archivo o cargar uno tienes que hacer lo mismo: abrir el archivo, hacer lo que *realmente* quieres hacer y luego cerrar el archivo. Esto es tedioso y fácil de olvidar. En Ruby, guardar (o cargar) archivos trabaja en forma similar al código anterior, entonces no tienes que preocuparte por nada más que por lo que quieres guardar(o cargar). (En el próximo capitulo mostraré donde encontrar información sobre guardar y cargar archivos.)

También pudes escribir métodos que determinan cuantas veces, o incluso *si* deben llamar a un proc. Aquí hay un método el cual podría llamar al proc la mitad de veces y otro el cual lo llamará el doble de veces:

Código:

```
def puedeHacerse unProc
  if rand(2) == 0
    unProc.call
  end
end

def hacerDosVeces unProc
  unProc.call
  unProc.call
end

parpadeo = Proc.new do
  puts '<parpadeo>'
end

mirada = Proc.new do
  puts '<mirada>'
end

puedeHacerse parpadeo
puedeHacerse mirada
hacerDosVeces parpadeo
```

```
hacerDosVeces parpadeo
hacerDosVeces mirada
```

Resultado:

```
<mirada>
<parpadeo>
<parpadeo>
<mirada>
<mirada>
```

(Si ejecutas el programa un par de veces, verás que la salida cambiará) Estos son algunos de los casos comunes de uso de procs lo que le permite hacer cosas, utilizando simplemente métodos no podriamos hacerlo. Seguramente, podrías escribir un método para que parpadee dos veces, pero no podrías escribir uno que haga *algo* dos veces!

Antes de continuar, vamos a ver un último ejemplo. Los procs que hemos visto son bastante similares. Es tiempo de ver algo diferente, entonces vamos a ver cuanto un método depende de un proc pasado a este. Nuestro método tomará algun objeto y un proc, y llamará a este proc sobre este objeto. Si el proc retorna falso, terminamos; en caso contrario llamaremos al proc con el objeto. Continuaremos haciendo esto hasta que el proc retorne falso (esto es mejor, o el programa finalizará con error). El método retornará el último valor no falso retornado por el proc.

Código:

```
def hacerHastaQueSeaFalso primeraentrada, unProc
  entrada  = primeraentrada
  salida = primeraentrada

  while salida
    entrada  = salida
    salida = unProc.call entrada
  end

  entrada
end

construirMatrizDeCuadrados = Proc.new do |array|
  ultimonumero = array.last
  if ultimonumero <= 0
    false
  else
    array.pop                          #  Quitar el último número...
    array.push ultimonumero*ultimonumero  #  ...y reemplazar este con el último número elevado al cuadrado...
    array.push ultimonumero-1           #  ...seguido por un número menor.
  end
end

siempreFalso = Proc.new do |soloIgnorame|
  false
end

puts hacerHastaQueSeaFalso([5], construirMatrizDeCuadrados).inspect
puts hacerHastaQueSeaFalso('Estoy escribiendo esto a las 3:00 am; ¡alguien que lo finalice!', siempreFalso)
```

Resultado:

```
[25, 16, 9, 4, 1, 0]
Estoy escribiendo esto a las 3:00 am, ¡alguien que lo finalice!
```

Está bien, este es un ejemplo bastante raro, debo admitirlo. Pero esto muestra como actúa diferente nuestro método cuando le damos diferentes procs.

El método `inspect` es muy parecido a `to_s` , salvo que la cadena que devuelve trata de mostrar el código ruby para crear el objeto que pasó. Aquí se nos muestra toda la matriz devuelta por nuestra primera llamada a `haceHastaQueSeaFalso`. Además, notamos que nosotros no procesamos el `0` al final de la matriz, porque `0` al cuadrado sigue siendo `0` y por lo tanto no tenía que hacerse. Y puesto que `siempreFalso` era siempre `false`,`hacerHastaQueSeaFalso` no hace nada la segunda vez que se llama sino que retorna lo que se le pasó.

Métodos que retornan Procedimientos

Una de las cosas interesantes que puedes hacer con procs es crearlos en los métodos y devolverlos. Esto permite realizar cosas grandiosas de programación (cosas con nombres impresionantes, como *lazy evaluation*, *estructuras de datos infinitas* y *currying*), pero el hecho es que casi nunca hago esto en la práctica, ni puedo recordar haber visto a nadie hacer esto en su código. Creo que es el tipo de cosas que no suelen llegar

a tener que hacer en Ruby, o tal vez simplemente Ruby te anima a buscar otras soluciones, yo no lo sé. En cualquier caso, sólo voy a referirme a esto brevemente.

En este ejemplo, `compose` toma dos procs y devuelve un proc nuevo que, cuando se le llama, llama al primer procy pasa el resultado en el segundo proc.

Código:

```
def compone proc1, proc2
  Proc.new do |x|
    proc2.call(proc1.call(x))
  end
end

cuadrado = Proc.new do |x|
  x * x
end

doble = Proc.new do |x|
  x + x
end

dobleYCuadrado = compone doble, cuadrado
cuadradoYDoble = compone cuadrado, doble

puts dobleYCuadrado.call(5)
puts cuadradoYDoble.call(5)

*Resultado:*

100
50
```

Ten en cuenta que la llamada a `proc1` tenía que estar dentro de los paréntesis para `proc2` con el fin de que se haga en primer lugar.

Pasando Bloques (no Procedimientos) en los Métodos

Ok, esto es académicamente interesante pero también algo difícil de usar. Gran parte del problema es que hay tres pasos que se tienen que realizar (definir el método, hacer el proc y llamar al método con el proc), parecería que solo debería haber dos (definir el método y pasar el *bloque* correcto dentro del método, sin necesidad de usar un proc) ya que la mayoría de las veces usted no desea utilizar el proc/bloque después de pasarlo al método. Bueno, no lo sabes, Ruby tiene todo resuelto por nosotros! De hecho, ya ha estabas haciendolo cada vez que utilizabas iteradores.

Te mostraré primero un ejemplo rápido, y luego vamos a hablar de ello.

Código:

```
class Array
  def cadaPar(&fueBloque_ahoraesProc)
    esPar = true  #  Empezamos con "true" porque las matrices comienzan con 0

    self.each do |objeto|
      if esPar
        fueBloque_ahoraesProc.call objeto
      end

      esPar = (not esPar)  #  Cambiar de pares a impares o viceversa
    end
  end

end

['manzana', 'manzana podrida', 'cereza', 'durian'].cadaPar do |fruta|
  puts '¡Yum!  Me encantan los pasteles de '+fruta+', ¿no?'
end

#  Recuerda,, estamos tratando de conseguir los numeros pares
#  de la Matriz.

[1, 2, 3, 4, 5].cadaPar do |bolaImpar|
  puts bolaImpar.to_s+' NO es un número par!'
end
```

Resultado:

```
¡Yum! Me encantan los pasteles de manzana, ¿no?
¡Yum! Me encanta pasteles de cereza, ¿no?
1 NO es un número par!
3 NO es un número par!
5 NO es un número par!
```

Así que para pasar un bloque de `cadaPar` todo lo que tenía que hacer era pegar el bloque después del método. Puedes pasar un bloque dentro de cualquier método de esta manera, aunque muchos métodos simplemente ignorarán el bloque. Con el fin de hacer que tu método *no* ignore el bloque debes apoderarse de él y convertirlo en un proc y poner el nombre del proc al final de la lista de parámetros de tu método precedida por el signo `&`. Así que esa parte es un poco difícil pero no demasiado y sólo tienes que hacer esto una vez (cuando se define el método). A continuación, puedes utilizar el método una y otra vez al igual que los métodos que reciben bloques como `each` y `times`. (Recuerda que con `5.times` ¿hacemos ...?)

Si estás confundido, sólo recuerda lo que `cadaPar` se supone que debe hacer: llamar al bloque pasado con todos los demás elementos de la matriz. Una vez que lo hayas escrito y funciona no es necesario pensar en lo que está haciendo en realidad internamente ("¿qué bloque se llama cuando?") De hecho, esto es exactamente por lo que escribimos métodos como éste: para que no tengamos que pensar de nuevo en cómo trabajan. Nos limitamos a usarlos.

Recuerdo una vez que quería ser capaz de medir la duración de distintas secciones de un programa. (Esto también se conoce como *profiling*.) Así que escribí un método que toma la hora antes de ejecutar el código, ejecuta y luego toma la hora al final para obtener la diferencia. No puedo encontrar el código en este momento, pero no lo necesito, ya que probablemente fue algo como esto:

Código:

```
def profile descripcionDeBloque, &bloque
  inicioHora = Time.now

  bloque.call

  duracion = Time.now - inicioHora

  puts descripcionDeBloque+':  '+duracion.to_s+' segundos'
end

profile '25000 duplicaciones' do
  numero = 1

  25000.times do
    numero = numero + numero
  end

  puts numero.to_s.length.to_s+' digitos'  #  El numero de digitos en este numero ENORME.
end

profile 'contar hasta un millon' do
  numero = 0

  1000000.times do
    numero = numero + 1
  end
end
```

Resultado:

```
7526 digitos
25000 duplicaciones:  0.246768 segundos
contar hasta un millon:  0.90245 segundos
```

¡Qué sencillo! Qué elegante! Con ese pequeño método puedo fácilmente saber cuanto tiempo demora parte de cualquier programa que quiero, solo ejecuto el código en un bloque y se lo envió a `profile`. ¿Qué podría ser más sencillo? En la mayoría de los lenguages, yo tendría que añadir explícitamente el código de tiempo (lo que está dentro de `profile`) dentro de cada sección que deseo medir. En Ruby, sin embargo, tengo que mantener todo en un solo lugar, y (más importante) ¡fuera de mi camino!

Algunas cosas por intentar

- *Reloj del Abuelo*. Escriba un método que toma un bloque y lo llame una vez por cada hora que ha pasado hoy. De esta manera, si paso al bloque `do puts 'DONG!' end` la campana debería sonar (más o menos) como un reloj de péndulo. Pon a prueba tu método con unas

pocos bloques (incluyendo la que acabo de darte). ***Sugerencia:*** Puede utilizar `Time.now.hour` para obtener la hora actual. Sin embargo, este devuelve un número entre `0` y `23`, por lo que tendrá que modificar los números a fin de obtener valores clásicos de un reloj de este tipo (`1` *al* `12`).

- *Program Logger*. Escribir un método llamado `log`, la cual toma una cadena de un bloque y, por supuesto, el bloque. Al igual que `doSelfImportantly`, deberá `puts` una cadena diciendo que se ha iniciado el bloque, y otra cadena diciendo que ha terminado el bloque y también debe decir lo que el bloque retornó. Pon a prueba tu método mediante el envío de un bloque de código. En el interior del bloque, pon *otra* llamada a `log` pasando otro bloque. (Esto se llama *anidación*.) En otras palabras, su salida debería ser algo como esto:

Listado:

```
A partir del "bloque externo" ...
A partir de "un bloque pequeño" ...
... "Algún pequeño bloque" terminó, regreso: 5
A partir del "otro bloque" ...
... "Otro bloque", terminó, regreso: me gusta la comida tailandesa!
... "Bloque exterior", terminó, regreso: false
```

- *Mejores Logger*. El resultado del último logger fue un poco difícil de leer y debería empeorar cuanto más se use. Sería mucho más fácil de leer si se indenta las líneas en los bloques interiores. Para ello, tendrás que llevar la cuenta de cuán profundamente estás anidado cada vez que el logger quiere escribir algo. Para ello, utilice una *variable global*, una variable que se puede ver en cualquier parte de tu código. Para hacer una variable global, sólo precede a su nombre con el símbolo `$` , como los siguientes: `$global`, `$nestingDepth` y `$bigTopPeeWee`. Al final, el logger debe generar un código como este:

Listado:

```
A partir del "bloque exterior" ...
  A partir de "un bloque pequeño" ...
    A partir del "pequeñito-minúsculo bloque" ...
    ... "pequeñito-minúsculo bloque" terminó, regreso: un montón de amor
  ... "un bloque pequeño" terminó, regreso: 42
  A partir del "otro bloque" ...
  ... "otro bloque", terminó, regreso: me encanta la comida india!
... "bloque exterior", terminó, regreso: true
```

Bueno, eso es todo lo que vas a aprender de este tutorial. ¡Felicitaciones! ¡Has aprendido un montón! Tal vez no tienes ganas de recordar todo, o te has saltado unas partes ... Realmente, eso está bien. La programación no es sobre lo que sabes, se trata de lo que puedes imaginar. Como siempre que se sepa dónde encontrar las cosas que habías olvidado, lo estás haciendo muy bien. ¡Espero que no pienses que escribí todo esto sin revisar estas cosas a cada minuto! Porque lo hice. Yo también recibí un montón de ayuda con el código de los ejemplos de este tutorial. Pero, ¿dónde estaba *yo* buscando estas cosas y donde *yo* pido ayuda?.

Te voy a enseñar...

Después de esta guía

Entonces, ¿qué haremos ahora? Si tienes una pregunta, ¿a quién consultarás? Que pasa si quieres un programa que abra una página web, envíe un correo electrónico, o redefina el tamaño de una foto digital? Bueno, hay muchos, muchos lugares donde encontrar ayuda Ruby. Desafortunadamente, esto es de poca ayuda, ¿no? :-)

Para mí, hay realmente solo tres lugares donde busco ayuda sobre Ruby. Si es una pequeña pregunta, y pienso que puedo experimentar yo mismo para encontrar la respuesta, utilizo *irb*. Si es una gran pregunta, busco por esto dentro de mis *herramientas*. Y si no puedo darme cuenta por mi mismo, entonces pregunto por ayuda en *ruby-talk*.

IRB: Ruby Interactivo

Si instalaste Ruby, entonces tienes instalado irb. Para usarlo, solo ve a consola y tipea `irb`. Cuando estes en irb, puedes tipear cualquier expresion ruby que quieras, y este te dirá el valor de esto. Tipea `1 + 2`, y este te devolverá `3`. (Puedes darte cuenta que no tienes que utilizar `puts`.) Esto es como una calculadora Ruby gigante. Cuando finalizes, solo debes tipear `exit`.

Hay un montón de cosas ademas para irb, pero puedes aprender mucho más probando esto.

El Pico: "Programming Ruby"

Absolutamente *el* libro Ruby a tener es "Programming Ruby 1.9, The Pragmatic Programmer's Guide", por Dave Thomas, Chad Fowler y Andrew Hunt (The Pragmatic Programmers). Mientras que recomiendo tomar la tercera edición de este excelente libro, con todo lo último cubierto.

Puedes encontrar todo lo que necesites de Ruby, desde lo básico a lo avanzado, en este libro. Es fácil de leer, fácil de entender, es perfecto. Desearía que cada lenguage tuviera un libro de esta calidad. Al reverso del libro, encontrarás una gran sección detallando cada método en cada clase, explicandolo y dando ejemplos. ¡Me encanta este libro!

Hay algunos lugares donde puedes conseguir una versión más antigua (incluyendo el sitio mismo Pragmatic Programmers), pero mi lugar favorito es en ruby-doc.org. Esta versión tiene una linda tabla de contenidos a un costado, como también un índice. (ruby-doc.org tiene un montón más de grandiosos contenidos también, como para el Core API y biblioteca estándar(Standard Library)... basicamente, documenta todo Ruby. Chequealo aquí.)

Y porque es llamado "¿el pico"? Bueno, hay una imágen de un pico en la tapa del libro. Es un nombre tonto, creo yo, pero quedó así.

Ruby-Talk: la lista de mails Ruby

Aun teniendo irb y el pico, algunas veces sigues sin resolver algo que necesitas. O quizás quieres saber si alguien ya hizo algo de lo tú estás intentando, para saber si puedes usar eso en cambio. En estos casos, el lugar para esto es ruby-talk, la lista de mails Ruby. Esta lleno de gente amigable, inteligente y con ganas de ayudar. Para aprender más de esto, o suscribirte, mira aquí.

ADVERTENCIA: Hay *muchos* emails en esta lista de correos cada día. Tuve que enviar directamente estos a diferentes carpetas de mail para que no me molesten. Si no quieres ocuparte de todos esos emails, pienso, no debes inscribirte. La lista de mails es espejada en el grupo de noticias comp.lang.ruby, y viceversa, entonces puedes ver los mensajes ahí. De igual manera, ves los mismos mensajes, pero en un formato diferente.

Tim Toady

Algo que traté de mostrarte, lo cual seguramente podrás probar pronto, es el concepto de TMTOWTDI (pronunciado como "Tim Toady" en inglés for *There's More Than One Way To Do It*): *Hay Más De Una Forma DE Hacer Esto*.

Algunos te dirán que TMTOWTDI es maravillos mientras otros puedan pensar algo diferente. No tengo fuertes sentimientos de las cosas en general, pero pienso que esto es una *terrible* forma de enseñar a programar a alguién. (Como si aprender a programar de una forma no fuera suficiente desafío y confuso!)

Sin embargo, ahora que te estás moviendose más allá de este tutorial, verás mucha más diversidad de código. Por ejemplo, pienso que al menos hay otras cinco formas de hacer un texto (aparte de las que encierran un texto en comillas), y cada uno de estos trabaja un poco diferente. Solo mostre lo más simple.

Cuando hablamos de ramificación, mostré el `if`, pero no mostré el `unless`. Voy a dejar que te des cuenta por ti mismo en irb de que se trata.

Otro lindo atajo que puedes usar con `if`, `unless` y `while`, es esta linda versión de una línea:

Código:

```
#  Estas lineas son de un programa que escribi para generar palabras
puts 'grobably combergearl kitatently thememberate' if 5 == 2**2 + 1**1
puts 'enlestrationshifter supposine follutify blace' unless 'Chris'.length == 5
```

Resultado:

```
grobably combergearl kitatently thememberate
```

Y finalmente, hay otro camino para escribir métodos la cual toma bloques (no procs). Vimos la parte donde tomabamos el bloque y lo convertíamos dentro de un proc usando el truco `&block` en la lista de parametros cuando defines una función. Entonces, para llamar el bloque, solo usas `block.call`. Bueno, hay una manera más corta (aunque personalmente pienso que es más confuso). En lugar de esto:

Código:

```
def hacerDosVeces(&block)
  block.call
  block.call
end
```

```
hacerDosVeces do
  puts 'murditivent flavitemphan siresent litics'
end
```

Resultado:

```
murditivent flavitemphan siresent litics
murditivent flavitemphan siresent litics
```

...haces esto:

Código:

```
def hacerDosVeces
  yield
  yield
end

hacerDosVeces do
  puts 'buritiate mustripe lablic acticise'
end
```

Resultado:

```
buritiate mustripe lablic acticise
buritiate mustripe lablic acticise
```

No sé... ¿qué piensas? Quizás solo sea yo, pero... ¡¿`yield`?! Si esto fue algo como `call_the_hidden_block` o otra cosa más, esto tendría *mucho más* sentido para mi. Mucha gente dice que `yield` tiene sentido para ellos. Pero, supongo que es algo como TMTOWTDI (Hay Más De Una Forma De Hacer Esto), y eso es todo: ellos hacen las cosas a su manera, y yo lo haré a mi manera.

Fin

Usa esto para lo bueno y no para lo malo. :-) Y si encuentras este tutorial útil (o confuso, o si encuentras un error), avísanos!